CHAPTER 1

1 INTRODUCTION

Computers are used in many areas today, for example: in education, multimedia, games, research etc. Using computers in their own languages is very difficult. So some kind of translation processes to the language of computer is needed. And *compiler* does that translation. The performance of the compiler can be increased in many ways like optimizations, using appropriate intermediate representation (IR), good memory management etc. IR is a vital part in designing compiler analyses and optimizations. Ultimately, the choice of IR for compiler is fundamental. Advances in IRs translate into advances in compilers. Understanding IRs in detail is important for achieving efficiency of compiler. Section 1.1 provides the motivation of this dissertation. Section 1.2 provides the brief introduction to the compiler. Translation Process is described in section 1.3. Section 1.4 describes about IR. Section 1.5 provides the analysis model. And finally, section 1.6 gives the outline of this dissertation.

1.1 Motivation

Today, there are many IRs that can be used in our compiler project. Selecting the right IR for compiler project requires an understanding of the source language, the target machine, properties of the source programs and strength and weakness of the programming language that the compiler will be implemented.

Each IR has its own advantages and disadvantages. Different IR has different style of representation. Source language influences the selection of IR. For example, if C is used as a source language then IR should be capable of resolving the pointers. Likewise, if C++ or Java is used as one of IR then IR should be capable of representing programs in object-oriented semantics. But putting the extra information in the IR may require extra cost. If the source language doesn't need pointers then additional information about the

pointers is not necessary. So the selection of IR requires consideration of the properties that must be performed on the IR and their costs and the range of constructs that must be expressed in the IR.

Some of the popular IRs are three-address code, abstract syntax tree (AST), control flow graph (CFG) [1], Static Single Assignment (SSA) [3], Program Dependence Graph (PDG) [6]. Recently other IRs like Static Single Information (SSI) [22], Value Dependence Graph (VDG) [24], Dependence Flow Graph (DFG) [25], Program Dependence Web (PDW) [26] are getting much attention in the research.

In the past, simple IRs like three-address code, AST, CFG were sufficient for source languages. As the new programming language and optimizations concept develop, and the machine become much faster, the need of appropriate IR becomes necessary. Selecting the right IR can simplify the translation from source program to intermediate form and help in further phases. So if there is some kind of comparative study that helps us to use suitable representation then it will reduce compiler design time as well as overall running cost.

1.2 Compiler

Compiler is a program that translates one form of language, the *source* language, into another, the *target* language [1]. If there are any errors in the source language the compiler gives the error messages to the user, see fig. 1.1. The output of the compiler is normally called the *object* language also which is close to the machine language of an actual computer. That object language could be either an assembly language of some variety of the machine language.

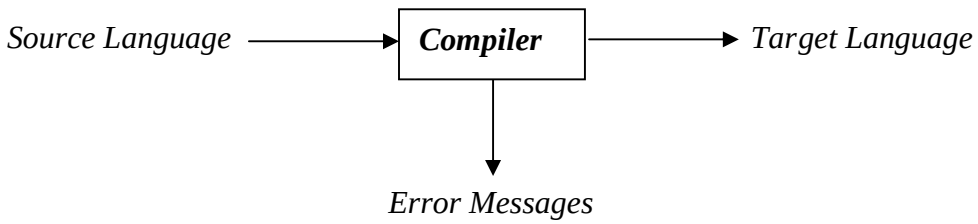

Figure. 1.1. A Compiler.

1.3 Translation Process

The translation process includes many phases. The translation may be very simple but the efficiency of the translated program may be not good. So some might use simple translation if they ignore efficient program. But today efficiency of program is such a big issue that the translation process is very complex. But broadly, they can be divided into two phases, shown in fig. 1.2. They are Analysis phase (front-end) and Synthesis phase (back-end).

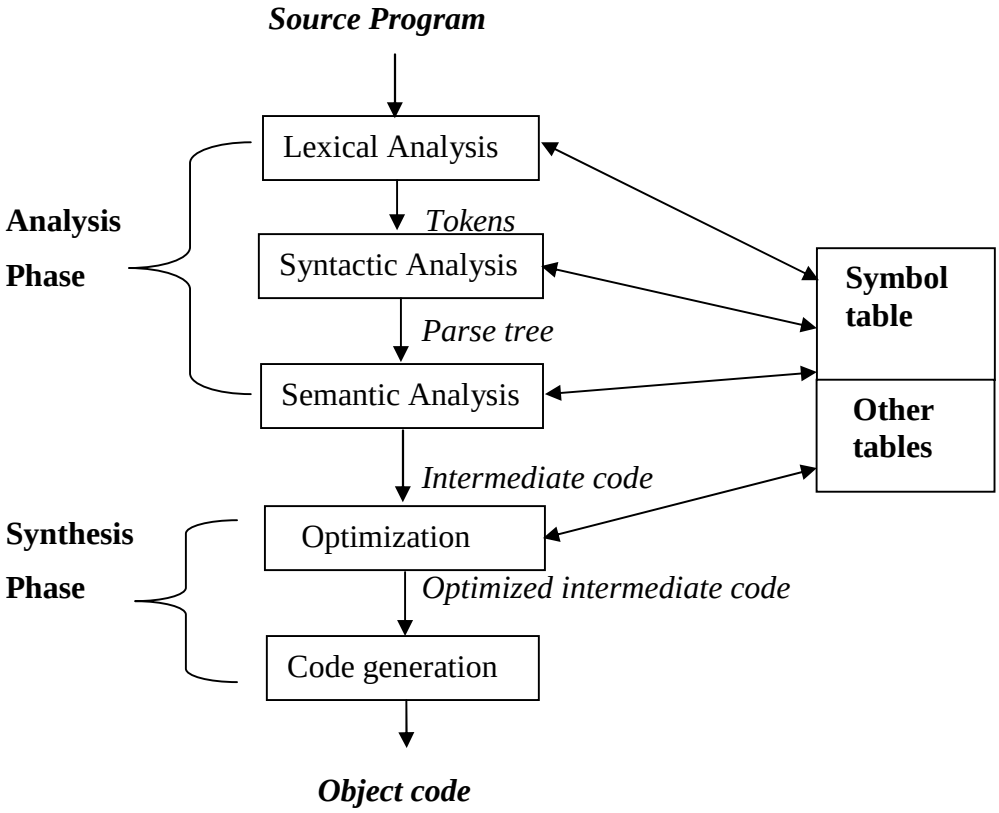

Figure 1.2. Phases of a Compiler.

Analysis phase includes lexical, syntax and semantic analyses. Synthesis phase includes optimization and code generation. Intermediate Representation (IR) of the source program is produced at the end of the analysis phase. IR is the representation of the source program and contains the necessary information so that it helps in further phases.

The IR can be divided into many levels: high-level, medium-level and low-level [2]. Intermediate representation serves as *bridge* between the analysis and synthesis phase. Various optimization algorithms can be applied in the IR. After that object code is generated in the code generation phase. And that object code is executed in the machine after linking (if needed) with other object codes. Intermediate representation of the program directly influences the synthesis phase and execution of the program.

Also the compiler can be grouped in the number of times it passes over the source program. A one-pass compiler, the source program is analyzed only once and translated directly to the object code. In this type, the compilation is fast but has slow execution speed. A two-pass compiler analyses the source program twice. The first pass extracts the informations, like variable name, from the source program. Then the second pass produces the object code from the extracted information.

1.3.1 Analysis Phase

At first, the source program is just a sequence of characters. So to move forward the compiler needs some analysis of the source program. The analysis includes Lexical analysis (scanning), syntactic analysis (parsing) and semantic analysis.

Lexical analyzer scans each character from the source program to group the sequence of characters to form the elementary constituents like: identifiers, keywords, operators, comments, blanks, and so on. The lexical analyzer analyses the source program to built basic program unit, called *tokens* (*lexical items*). These tokens are then the input for next stages.

The next stage is the syntactic analysis or parsing. In this stage, the tokens that are produced by the lexical analyzer are grouped together to form the larger program structures like: expressions, statements, subprogram call or declaration. It uses the formal grammar to construct the larger program structures.

The last stage in the analysis phase is the semantic analysis stage. Here the semantics of the program structures are analyzed. The semantic analyzer is important stage in the translation process. Its work includes symbol table maintenance, error detection etc. It produces the program in some kind of internal form i.e. intermediate representation. And that IR is used in next phase.

1.3.2 Synthesis Phase

After representing the source program in intermediate form, the translation process enters into the synthesis phase. This phase includes optimization and code generation.
The semantic analyzer produces the intermediate code for the code generation. But that code may be very inefficient. So some optimizations are applied before the code generation.

The final code is generated in the code generation stage. The final code may be machine language, assembly code or object code. This stage produces the output of the translation process.

If the source program is translated in many subprograms then the output of these subprograms need to link with each other to form a single output object code. This stage includes work like data references, internal procedure calls etc.

1.4 Intermediate Representation (IR)

IRs are very important in the compiler technology. There are many IRs today. Different IRs has different styles and advantages although there are some common advantages. They are as below:

1) Portability of program: retargeting at different machine architectures
2) Optimizations: doing optimizations in the IR so that the programs could run faster than normal programs.

Fig. 1.3 gives the possible place of the IR in the translation process.

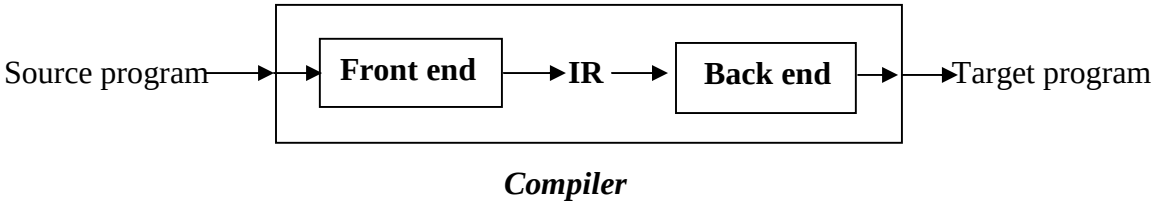

Figure 1.3. Position of IR in Compiler.

This thesis is all about learning various IRs and selecting the appropriate IR for compiler.

1.5 Comparative Analysis

After studying many IRs, this study has presented comparative analysis for selecting the right IR for the compiler that is efficient in terms of construction time, space and execution speed.

1.6 Outline of the Thesis

The rest of this dissertation is organized as follows. Chapter 2 presents the brief discussion of the background and literature review of the various IRs and their taxonomy. This chapter also gives the brief introduction to the tools used in this thesis and the motivation of this thesis work. Chapter 3 describes the selected IRs for our implementation and tools used in implementation in greater detail. In particular, this study has selected Static Single Assignment (SSA) and Program Dependence Graph (PDG). SUIF and Machine SUIF compiler infrastructures are used for implementation. Chapter 5 describes the implementation detail of this study. Chapter 4 consists the testing of results that are obtained in this study. Finally, Chapter 5 concludes the dissertation and summarizes the main insights gained through this work.

CHAPTER 2

2 BACKGROUND

This chapter presents the brief discussion about the various IRs that are developed and used over the last 50 years in the compiler technology. Those IRs are divided and discussed briefly so that the understanding of the IR would be better. Section 2.1 gives the literature review of various IRs and their taxonomy. Finally, section 2.2 contains the tools available and used in this thesis.

2.1 Literature Review

Since there are many IRs that are developed and used in compiler in history, this study will consider only those IRs that are relevant to this thesis. Overall the IRs can be divided into three groups:

1) Sequential IR,
2) Tree-based IR, &
3) Graph-based IR.

2.1.1 Sequential IR

This type of IR is similar to assembly language. The three address code [1] is a good example of sequential IR. It is a sequence of statements of the general form: x = y op z, where, x, y, & z are names, constants or compiler generated temporaries; op is any operator or a logical operator on Boolean valued data. It is a linearized representation of a syntax tree of DAG [1, 2] in which explicit names correspond to the interior nodes of the graph. The three-address code can be implemented as many form: Quadruples, Triples and Indirect Triples [1]. Muchnick's MIR [2] is similar to three address code.

2.1.2 Tree-based IR

The most common IR is the Abstract Syntax Tree (AST). It makes the structure of a program. A major use of AST is in language-sensitive or syntax-directed editors for programming languages, in which they usually are the standard internal representation for programs [2]. For example consider the assignment statement: x = y + z * y + z;. The AST would be as fig. 2.1:

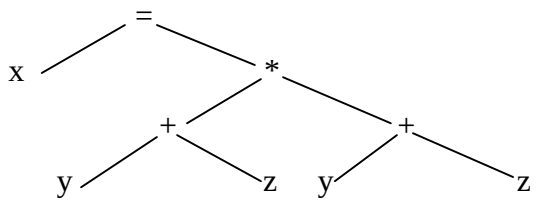

Figure 2.1. Abstract Syntax Tree.

2.1.3 Graph-based IR

a) Directed Acyclic Graph (DAG): DAG can be used as the IR in which each node represents the basic block. The DAG representation of a basic block is as the compressing the minimal sequence of trees that represents it still further. The leaves represent the values of the variables and constants available on entry to the block that are used within it. The other nodes of the DAG all represent operations and may also be annotated with variable names, indicating values computed in the basic block. The main advantage of the DAG is that it reuses values, and so is generally a more compact representation than trees or the linear notations [2]. For example consider the previous assignment x = y +z * y+ z; , then the DAG would appear as fig. 2.2.

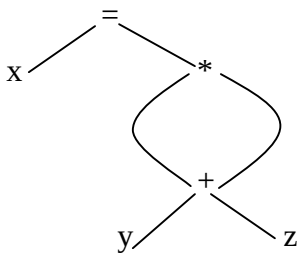

Figure 2.2. Directed Acyclic Graph.

b) Control Flow Graph (CFG): CFG is the most important IR that is used in the compiler. The information found in the CFG is used by various other IRs like: Static Single Assignment (SSA), Static Single Information (SSI), Program Dependence Graph (PDG) and so on. In CFG, each node is a basic block. Each basic block contains one or more sequence of instructions. Those instructions may be three-address code or assembly code. Directed edges are used to represent the flow of control. The flow of control may be fall through, conditional jumps, unconditional jumps, and so on.

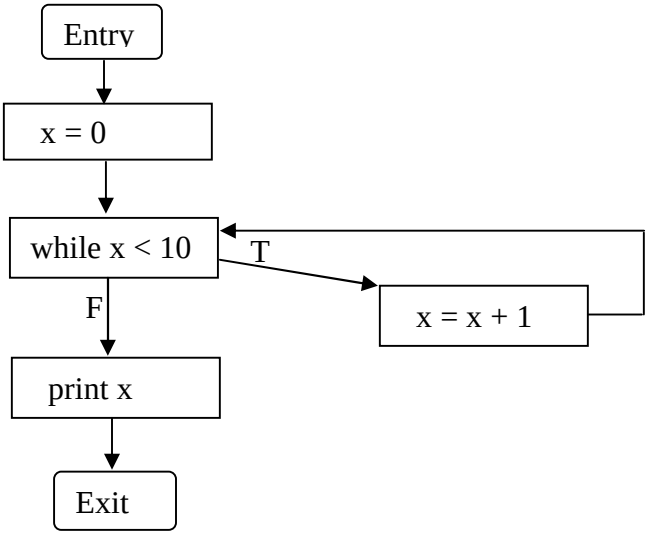

Figure 2.3. Control Flow Graph

There are, in most representations, two specially designated blocks: the entry block, through which control enters into the flow graph and the exit block, through which all control flow leaves. When these two special blocks are added to the normal CFG, it is

called augmented CFG, fig. 2.3. CFG is essential to many optimizations and static tools. It is used in detecting reachability of the code and other optimizations like constant propagation [17], constant folding. More details about the CFG can be found in Aho et al. [1], Muchnick [2].

c) Data dependence graph (DDG): DDG is an important IR. It is also called Data flow graph. It is different from the CFG because the main theme of the DDG is data flow oriented [1]. Nodes in the DDG are similar to the nodes of the CFG. But the meanings of directed edges are totally different to the meaning of those found in the CFG. Each edge from one node to another represents the dependence of the information or data between the two nodes. For example if (I ,J) is any edge in the DDG, then the information found in the node I is needed in the node J. Node J can not be executed before node I. Fig. 2.4 gives the clear idea about the DDG.

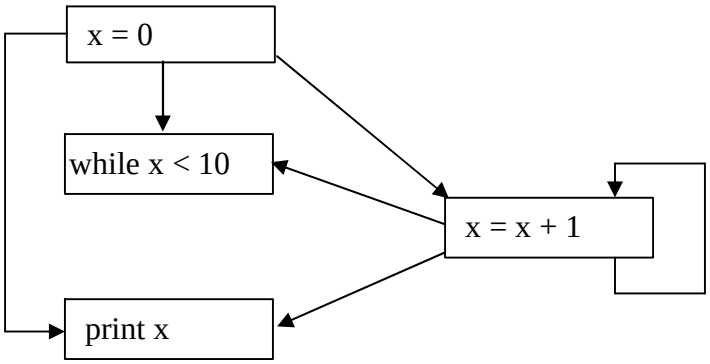

Figure 2.4. Data Dependence Graph.

e) Static Single Assignment (SSA): Cytron et al. present SSA form [3, 4]. It also shows how to compute the SSA efficiently. It introduces a new structure called dominance frontiers. It shows how to compute SSA and the control dependence graph efficiently using dominance frontiers. The method presented behaves linearly with respect to program size for programs restricted to certain control structures. [2,3] use the SSA form. SSA form is the extended form of the CFG. In SSA, a variable is assigned only once. The SSA form has two properties: Each programmer-specified use of a variable is reached by

exactly one assignment to that variable. The program contains phi-functions that distinguish values of variables transmitted on distinct incoming control flow edges. [15] used SSA for efficiently determining the determining the equality of variables in programs. [16] used SSA for redundant computation using Global value numbering. SSA can be used to find Constant propagation [17].

Briggs et al. [5] present efficient way of constructing and destruction of the SSA form. This paper addresses three problems that arise in the use of the SSA form. The two solutions are the improvements to the construction of the SSA form from Cytron et al. [3]. It builds minimal, pruned and semi-pruned SSA form. And the last problem that this paper addresses is the process of converting SSA form back into executable code.

Cliff click et al. describes a simple graph-based intermediate representation [23]. Here the IR is basically the SSA but it implements the phi-functions for control dependences. Data dependences are represented using use-def edges and control dependences become edges to REGION nodes [23]. SSA will be described in detail in chapter 3.

f) Static Single Information (SSI): Ananian [22] presents SSI that can be used as IR for the compiler. SSI extends SSA form. SSI is similar to the SSA but it introduces more definitions. SSI form recognizes that information about variables is generated at branches and generates new names at these points. This provides us with a one-to-one mapping between variable names and information about the variables at each point in the program. Analyses can then associate information with variable names and propagate this information [22].

Formal Definition of SSI [22]: Pseudo-assignments are added for a variable V:

(ϕ) at a control-flow merge when disjoint paths from a conditional branch come together and at least one of the paths contains a definition of Vi and

(σ) at locations where control-flow splits and at least one of the disjoint paths from the split uses the value of V.

Details about SSI can be found in [18,22].

g) Program Dependence Graph (PDG): Ferrante et al. [6] present the PDG. The PDG consists Control-dependence graph (CDG) and Data-dependence graph (DDG). The nodes are statements and predicate expressions (or operators and operands) and the edges incident to a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends [6]. That is there are two types of edge found in the PDG. One is control dependence edge and other is data dependence edge. The PDG can be used not only for the optimizing compilers but it can be used in parallel machine. The paper describes how the PDG can be used for vectorization, node splitting, code motion, and loop fusion [6]. PDG will be discussed in more detail in chapter 3.

h) Value Dependence Graph (VDG): Weise et al. [24] present Value Dependence Graph. VDG is a functional IR that expresses computation solely as value flow. CFG-based IRs are statement based and name all values. PDG do the same. In contrast, a VDG program only specifies the flow of values through a computation. There is no superfluous information concerning names, or the order in which values are computed. (In effect, VDG edges correspond to uses of CFG virtual register names.) VDG has a demand-based semantics, so a value is only computed if it is needed by another computation [24].

i) Dependence Flow Graph (DFG): DFG is presented in Pingali et al. [25]. DFG is an improved version of def-use chaining. DFG has explicit control flow edges, and explicit def-use edges, but def-use edges are factored at control flow split and merge points, which reduces the expense of def-use chaining.

j) Program Dependence Web (PDW): PDW is a program representation that is suitable for control-driven, data-driven, or demand-driven interpretation [26]. Program Dependence Webs are an extension of PDG [26] and SSA form [3,4]. PDW provides a basis for program optimizations and Robert et al. present and efficient methos for translating an imperative program into a PDW [26].

2.2 Tools

2.2.1 Stanford University Intermediate Format (SUIF)

The SUIF system is a compiler infrastructure. It supports collaborative research and development of compilation techniques, based upon a program representation [31]. SUIF provides useful abstractions and frameworks for developing new compiler passes. Two versions of SUIF systems are found. One is SUIF1 and another is SUIF2. The design and implementation of SUIF1 is totally different from the SUIF2 system [31]. In this dissertation SUIF2 compiler infrastructure is used for our implementation. The key features of SUIF2 are:

1 It works in modular subsystem. User can add their own components, program representations and program analyses, easily.

2 An extensible program representation that allows users to create new instructions to capture new program construct semantics or new program analysis concepts.

More information can be found in SUIF home page[1]

2.2.2 Machine SUIF

Machine SUIF [32] is a research compiler infrastructure, developed at Harvard University, that is flexible, extensible, and easily-understood for constructing compiler back ends. In Machine SUIF, the optimization and analysis passes are coded in such a way that makes them as independent of the compiler environment and compilation targets as possible . And that is their main philosophy.

The Machine SUIF works under the SUIF [31] compiler infrastructure (version 2.1). The Machine SUIF provides an interface called, Optimization Programming Interface (OPI).

[1] See URL, http://suif.stanford.edu/

Machine SUIF is used for implementation and described in detail in chapter 4. To see more about Machine SUIF go to the home page.[2]

2.2.3 LCC

LCC is an easily retargetable ANSI C compiler written by Christopher Fraser and David Hanson. The front end eas adapted for use in the SUIF system. The compiler contains no optimizer, but its intermediate code is suitable for most of the optimizations.

2.2.4 Visualization for Compiler Graphs (VCG)

The VCG [33] is a tool that is used for the visualization for compiler graphs. It is developed by George Sander and Iris Lemke at the University of Saarland, Germany. The VCG tool reads a VCG specification and visualizes the graph. This work has used VCG 1.30 although there are few versions of the VCG available. The Machine SUIF provides interface for VCG and with that specific specification we can see the graph in X11. Detail description about VCG can be found in this page[3].

[2] See URL, http://www.eecs.harvard.edu/machsuif

[3] See URL, http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html

CHAPTER 3

3 TOOLS AND INTERMEDIATE REPRESENTATIONS' ANALYSIS

In the last chapter, various IRs and some tools that are relevant to this dissertation are presented in briefly. This chapter describes the selected IRs and the tools used to implement those IRs in greater detail. Section 3.1 gives the description of SUIF [31] system. Section 3.2 describes about the Machine SUIF [32] and its main applications in this thesis. Section 3.3 provides an overview of Static Single Assignment (SSA) [3, 4], its variants and construction methods for those variants of SSA. Finally, section 3.4 provides overview of Program Dependence Graph (PDG) [6] and its properties.

3.1 Stanford University Intermediate Format (SUIF)

The SUIF system is a compiler infrastructure designed to support collaborative research and development of compilation techniques, based upon a program representation [31]. Users are provided useful abstractions and frameworks for developing new compiler passes and by providing an environment that allows compiler passes to easily inter-operate so that maximum code could be reused. There are two versions of SUIF, SUIF1 and SUIF2 but this work has used SUIF2. The SUIF2 system was is built as part of the National Compiler Infrastructure (NCI) project The SUIF2 system is new design and implementation, and is completely different from the SUIF1 system.

The SUIF1 system was originally designed to support high-level program analysis of C and FORTRAN programs. But SUIF2 was designed to support the planned components such as object-oriented programming languages and better machine-level optimizations. SUIF2 supports new research topics that have yet to be defined as much as possible. SUIF1 system is a series of standalone program passes, that makes it inefficient when each time the program is read and write to disk for each passes. SUIF2 was developed with modular system that enables components to interoperate in a flexible manner.

3.1.1 Key Features of SUIF System

SUIF2 was designed to meet the goals of a research compiler infrastructure with the following key features:

1 different components like program representations and program analyses can be combined easily. That is, it works in a modular subsystem. Programmers can combine many modules with a driver to produce a standalone program. A compiler may be either a series of standalone programs that read and write SUIF file or a program that dynamically imports and applies a series of different modules to the program in memory.

2 users can extend the exiting program representation so that the new semantics of the program or new analysis concepts could be developed. Users can extend the object hierarchy that capture the program semantics to their needs but the SUIF program will always contain the same basic information but may contain different subsets of nodes representing with refined program semantics.

3.1.2 The SUIF Architecture

The SUIF system has a simple and modular architecture [31]. It contains three components: *kernel, modules* and *driver.* The kernel implements a set of basic functions found to be useful across all compilation passes. A number of modules loaded dynamically under user control. And driver controls the system operation. Figure 3.1 shows the SUIF architecture.

a) The Kernel

The kernel contains two layers. One is the IO kernel and other is the SUIF kernel. The IO kernel implements a persistent object system that is independent of the applications of writing compilers. The SUIF kernel defines and implements the Suif compiler environment (SuifEnv) that is all the user needs to know when writing a SUIF program. The SuifEnv is the only environment that holds the entire state and components of the compiler system, there are no other global variables and states in the system. To start a SUIF program, the instance of SuifEnv object must be created.

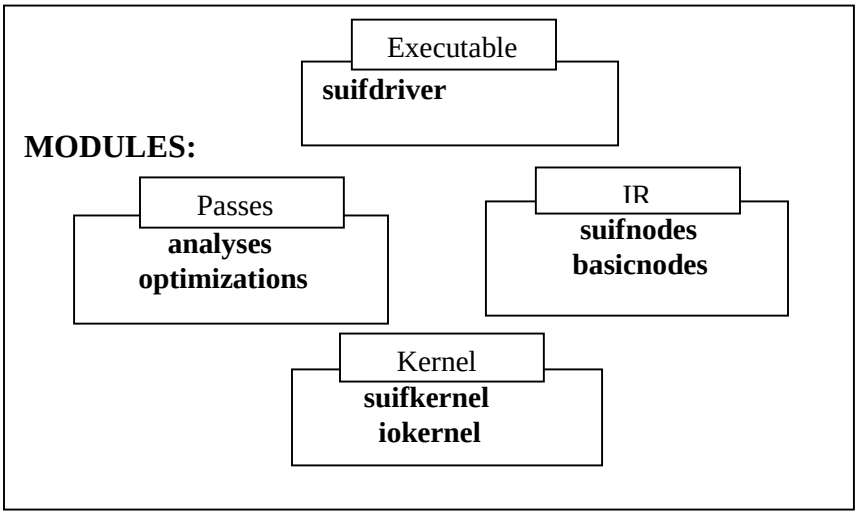

Figure 3.1. The SUIF system architecture.

The SuifObject is made up of the following components:

1. the program representation, which is the SUIF IR of the program currently stored in the SUIF environment.
2. the object factory, which is used by the system internally to create all persistent objects in the environment.
3. the subsystems, which have distinct duties like printing node information, printing error information, cloning of trees, dynamically loading SUIF programs, and the initialization and registration of modules.

b) Modules

The bulk of the SUIF compiler system is structured as modules**,** each of which is a C++ class identified by a unique module name. The system comes with a number of basic modules, as well as some tools to help user construct their own modules. Modules can be one of two kinds:

1 Intermediate Representation, which contains a set of nodes. The set of nodes contains suifnodes and basicnodes. The suifnodes captures standard programming constructs in standard languages such as C and FORTRAN. The basicnodes contains a number

of basic programming constructs.

2 Passes, which could be analyses or optimizations. The SUIF infrastructure contains many basic modules such as loading and printing a SUIF program that can be derived by users to define their own passes.

c) Drivers

The user needs to supply a "main" program that creates the SuifEnv, imports the relevant modules, loads a SUIF program and applies a series of transformations on the program and eventually writes out the information to create a compiler or a standalone pass in SUIF system. Suifdriver is one such driver that allows the user to dynamically specify the components and passes applied.

3.2 Machine SUIF

Machine SUIF is a flexible, extensible, and easily-understood infrastructure for constructing compiler back ends that is developed at Harvard University [32]. The main theme of Machine SUIF is to code analysis and optimization passes in such a way that they are as independent of the compiler environment and compilation targets as possible.

The Machine SUIF contains the Stanford SUIF [31] compiler infrastructure (version 2.1) for the compiler environment. But the Machine SUIF can use other compilers for the compiler environment. The SUIF compiler is capable of compiling C and FORTRAN code and produce optimized code for machine based on the Alpha or x86 architectures. The analyses and optimizations distributed in Machine SUIF do not directly reference any SUIF constructs or embed constants from any target machine. Instead, each is written using a standardized view of the underlying compiler environment, an interface layer called *Optimization Programming Interface (OPI)*.

3.2.1 Goals of Machine SUIF

There are three primary goals of Machine SUIF. They are: ease of use, quality of

optimized code, and reuse of optimized code.

a) Ease of use

This is the first and foremost goal of Machine SUIF. The Machine SUIF had to be easy, especially if user want to do something simple, and straightforward to retarget so that the user could use it in architecture investigations. Most of the users want to add or change the system in some way. Machine SUIF provides many libraries that can be easily used in user's pass for new analysis and optimization passes. It also supports parameterized passes. Target specific or environment specifics could be given to user's algorithm for analyses and optimizations. The OPI also hides the implementation of IR, many details of which are uninteresting to someone simply wanting to code a new optimization algorithm.

b) Quality of optimized code

Machine SUIF is a powerful system for producing optimized code. It is modular, making it easy to add, remove, and rearrange passes. User typically develops optimization passes that focus on a single action, such as dead code elimination. Nearly all such passes can be inserted at any point in the back-end flow. Thus, the writer of copy-propagation pass can focus on data flow and operand rewriting and assume that a subsequent run of the dead code elimination pass will remove any dead code produced during copy propagation.

c) Reuse of optimized code

Finally, Machine SUIF has to build in a manner that permitted reuse of existing optimizations directly in an optimization environment with significantly different constraints.

3.3 Static Single Assignment (SSA)

SSA form is an intermediate representation that can be used in compilers for various program analyses and optimizations which is developed Cytron at el. [3, 4]. Since the introduction of SSA form in 1989 [4], there are many analyses and optimizations that are based on the SSA form [15,16,17]. It has become a popular representation for use in data-flow analysis and optimization. In compilers, programs are translated into SSA form, perform analyses and optimized in many ways and then translated back out of SSA form. Figure 3.2 shows this process.

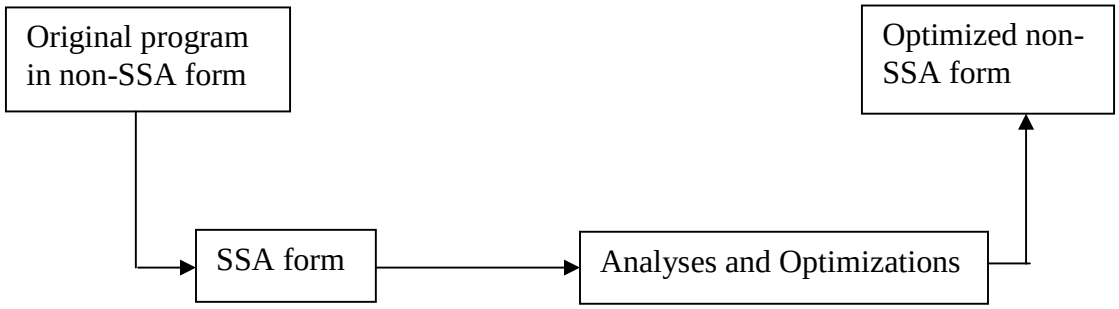

Figure 3.2. Role of SSA form in compilers.

3.3.1 Definition of SSA

In SSA form each variable is assigned only once statically. A single program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text [3]. SSA form can be viewed as a sparse representation of the use-def or def-use chains.

A program is translated into SSA form in two-step. In the first step, some trivial ϕ-functions are inserted at the some of the join nodes in the program's control flow graph. The ϕ-function has the form $x \leftarrow \phi(x, x, \ldots)$. In the next step, new variables are generated for each variable in the original program. Generally, new variables are the subscripted version of the variable that is found in the original program. In a ϕ-function, there may be

many arguments. These arguments are the variables. The number of operands is the number of control flow predecessors of the node that contains the ϕ-function. The jth operand of ϕ-function is associated with the jth predecessor of that node. When the control is reached to node N through its jth predecessor, then the ϕ-function contain in N will take jth operand from the operands. Translation to SSA form replaces the original program by a new program with the same control flow graph.

Figure 3.3 shows a simple control flow graph and its SSA form.

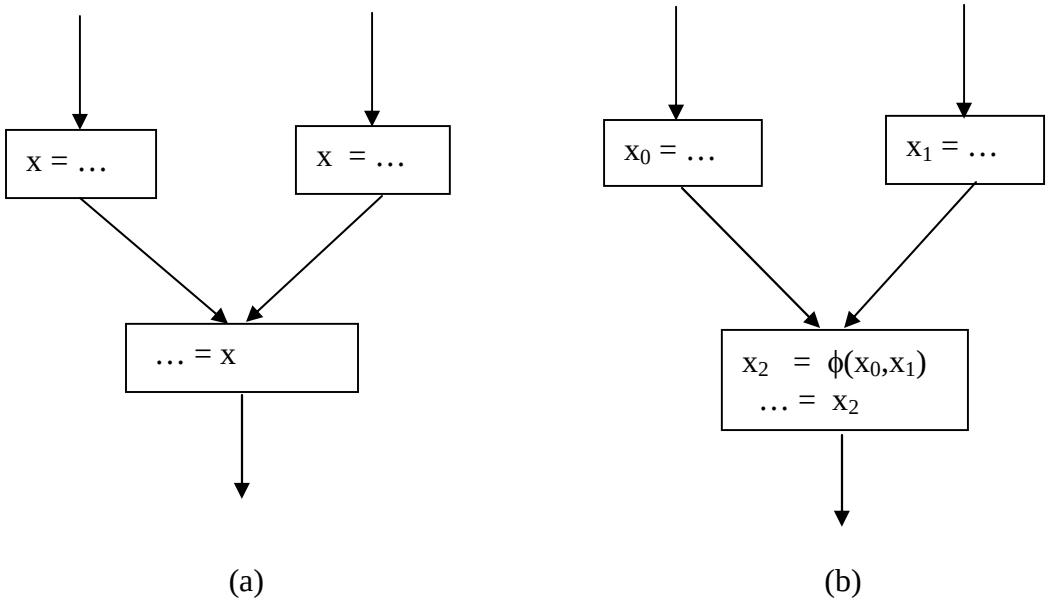

Figure 3.3. a simple control flow graph (a) and its SSA form (b).

Formally, a program is in SSA form if it follows following three conditions [3] :

1 If two nonnull paths X→Z and Y→Z converge at a node Z, and nodes X and Y contain assignments to V (in the original program), then a trivial ϕ-function V←ϕ(V,…,V) has been inserted at Z (in the new program).

2 Each mention of V in the original program or in an inserted ϕ-function has been replaced by a mention of a new variable Vi, leaving the new program in SSA form.

3 Along any control flow path, consider any use of a variable V (in the original program) and the corresponding use of V_i, (in the new program). Then V and V_i have the same value.

There are three flavors found in the SSA form. The SSA form that is constructed by these algorithms differs in the cost of construction, number of ϕ-functions that is inserted and the size of the name space. The three flavors are:

1) Minimal SSA,
2) Pruned SSA, and
3) Semi-pruned.

The simplest algorithm for translating the program into the SSA form is the minimal SSA form. This algorithm would insert ϕ-function at each join nodes in the control flow graph. Then the renaming process takes place. Even though the minimal SSA form inset few extra ϕ-functions the resulting SSA form is still valid. However, theses extra ϕ-functions causes overhead in analyses and optimizations. The other two variants of SSA form are similar to the minimal SSA form. In pruned SSA form, ϕ-function is inserted at join node N only if the variable defined is live in node Z or after the node Z. That is, live analysis is needed to translate into the pruned SSA form. There are fewer ϕ-nodes found in the pruned SSA form but the translation cost is much higher that the minimal SSA form. The third and last variant of SSA form is semi-pruned SSA form. The semi-pruned SSA form is developed by Briggs el at. [5]. There are fewer ϕ-nodes than the minimal form without the expense of solving data-flow equations to determine which values are “live”. The three variants of SSA form will be described in detail in the subsequent sections. But before that some important definitions are needed that are used in the construction of SSA form.

3.3.2 Dominance

Dominance relation between nodes in the control flow graph is important for the construction of SSA form.

3.3.3 Dominator Trees

In a control flow graph CFG, if node X appears on every path from the start node to node Y, then X *dominates* Y. Dominance is denoted by $\underline{\gg}$. If X dominates Y and $X \neq Y$, then X *strictly dominates* Y. Strict dominance is denoted by $\gg$. If X does not strict dominate Y, we write $\not\gg$. The *immediate dominator* of Y (*idom*(Y)) is the closest strict dominator of Y [3]. In *dominator tree,* the parent of each node is its immediate dominator. All the nodes that dominate the node X are the ancestors in the dominator tree. Lengauer and Tarjan give an efficient algorithm for building the dominator tree in **O**(ElogN) time, where E is the number of edges and N is the number of blocks in the CFG [12].

3.3.4 Dominance Frontiers

The *dominance frontiers* of a CFG node X is the set of nodes Y such that X dominates a predecessor of Y, but X does not strictly dominate Y [3]. It is denoted by DF(X):

$$DF(X) = \{ Y \mid (\exists P \in Pred(Y))(X \underline{\gg} P \text{ and } X \not\gg Y) \}$$

The computation of dominance frontier from the definition would be very inefficient. So Cytron et al. define two intermediate sets DF_{local} and DF_{up} for each node such that the following equation holds [3]:

$$DF(X) = DF_{local}(X) \cup \coprod_{Z \in Children(X)} DF_{up}(Z)$$

The DF_{local} (X) is defined by [3]:

$$DF_{local} (X) \overset{def}{=} \{ Y \in Succ(Z) \mid X \not\gg Y\}$$

Given any node Z that is not the root of the dominator tree, some of the nodes in DF(Z) may contribute to DF(*idom*(Z)). The contribution DFup(Z) that Z passes up to *idom*(Z) is defines by [3]:

$$DF_{up}(X) \stackrel{def}{=} \{ Y \in DF(Z) \mid idom(Z) \not\gg Y \}$$

An algorithm for finding dominance frontiers which runs in **O**(E + N2) time is developed by Cytron et al. [3]. Cytron et al. show that the algorithm is correct. This algorithm will be used in the implementation for finding SSA form and will be described in chapter 4 in more detail.

3.3.5 Relation between Dominance Frontiers and Joins

Cytron et al. extend the concept of dominance frontier and show how to place ϕ-functions in the nodes of CFG [3]. If S is a set of CFG nodes, then the set *join* nodes, denoted by J(S), is defined to be the set of all nodes Z such that there are two nonnull CFG paths that start at two distinct nodes in S and converge at Z. The *iterated* join $J^+(S)$ is the limit of the increasing sequence of sets of nodes

$$J_1 = J(S)$$

$$J_{i+1} = J(S \cup J_i)$$

In particular, if S happens to be the set of assignment nodes for a variable V, then $J^+(S)$ is the set of ϕ-function nodes for V.

The dominance frontier of a set of nodes is defined to be the set of nodes in the dominance frontier of any member of the set:

$$DF(S) = \bigcup_{X \in S} DF(X)$$

The *iterated* dominance frontier $DF^+(S)$ is the limit of the increasing sequence of sets of nodes

$$DF_1 = DF(S)$$

$$Df_{i+1} = DF(S \cup DF_i)$$

If the set S is the set of assignment nodes for a variable V, then Cytron et al. show that the iterated join node $J^+(S)$ is equal to iterated dominance frontier $DF^+(S)$. That is:

$$J^+(S) = DF^+(S)$$

This means that ϕ-nodes for variable V are required only in blocks in $DF^+(S)$. The insertion of ϕ-node can be efficiently done by other schemes [5,8,9].

3.3.6 Minimal SSA form

The algorithm for constructing minimal SSA form from program in CFG form requires two steps [3]. Figure 3.4 shows the algorithm for constructing minimal SSA form.

```
/* STEP 1: Insertion of ϕ-nodes */
Calculate the dominator tree and dominance frontier

For each variable V
    Calculate set of CFG nodes S which contain assignment to V
    Insert ϕ-node for V in the iterated dominance frontier of DF+( S )

/* STEP 2: Renaming each variable */
   Rename each variable by preorder walk over the dominator tree
```

Figure 3.4. Algorithm for building minimal SSA form.

The first step includes insertion of ϕ-nodes for each variable V in the iterated dominance frontier of $DF^+(S)$, where S is the set of CFG nodes that contains assignment to variable V. In the second step, renaming of variables take place. Detail implementation of this algorithm will be presented in chapter 4.

3.3.7 Pruned SSA form

In minimal SSA form, insertion of ϕ-node takes place according to the calculation of dominance frontier that correctly captures the potential flow of values. But because of the ignorance of the data-flow information, the minimal SSA form construction will insert a ϕ-node for variable V at a join point where V is not live.

Cytron et al. provide another variation of SSA form that they called *pruned* SSA form [3]. The construction of pruned SSA form needs "live analysis" first. A variable is *live* at a particular point in a program if there is a path to the exit along which its value may be used before it is redefined. It is *dead* if there is no such path. That is, for each block, a set of values that are live on entry to the block and can be referenced along some path leading to the block [1, 2].

Constructing pruned SSA form needs some changes to the algorithm given in fig. 3.3. In fig. 3.3, ϕ-node is inserted in join nodes but in pruned SSA form, liveness analysis is performed first. So the first step in the construction of pruned SSA form is the insertion of ϕ-node for V is inserted in every node $n \in DF^{+}(S)$, where $V \in$ Liveness(n) and S is set of CFG nodes that contain assignment to V. Pruned SSA from contains much less ϕ-nodes than minimal SSA form. The renaming process in pruned SSA form is same as in minimal SSA from.

The construction cost of pruned SSA form is much higher than minimal SSA form because the pruned SSA form must do liveness analysis before insertion of ϕ-node. Although linear-time or near-linear time algorithm exist for insertion of ϕ-nodes larger memory requirements can directly degrade performance.

3.3.8 Semi-Pruned SSA form

As described in previous sections, the SSA variants developed by Cytron et al. vary in the number of insertion of ϕ-nodes. Minimal SSA form inserts ϕ-function according to the calculation of dominance frontier without liveness analysis. Pruned SSA form inserts ϕ-nodes after calculation of liveness analysis and dominance frontier. Even though pruned SSA form inserts less ϕ-nodes than minimal SSA form, the construction cost of pruned SSA form is very high.

Briggs et al. developed a third variant of SSA that they called *semi-pruned* SSA form [5]. The main theme of semi-pruned SSA form is the speed and space advantage over the

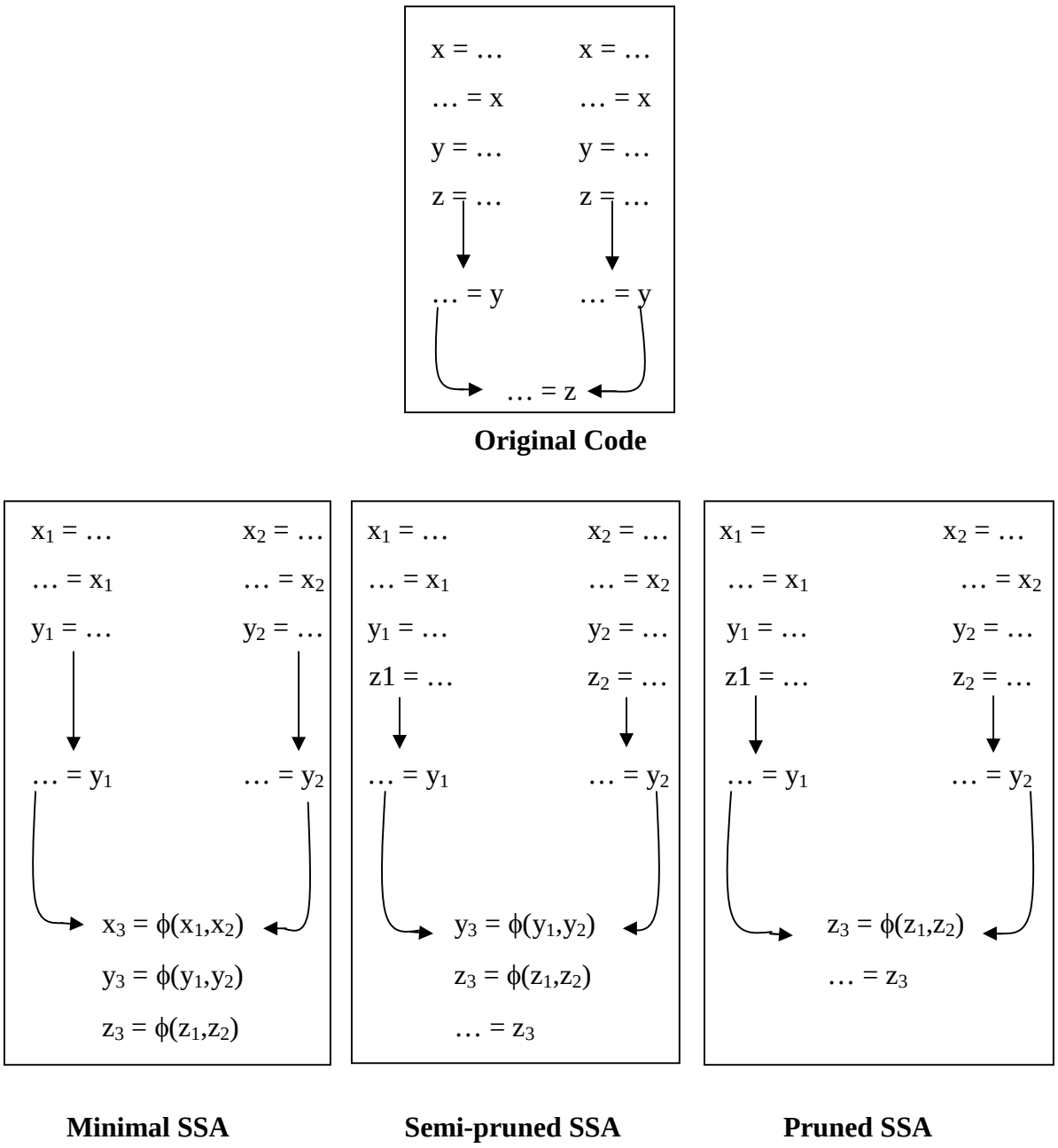

Figure 3.5. Three flavors of SSA form.

other two relies on the observation that many names in a routine are defined and used wholly within a single basic block [5]. In general, the compiler generates the temporary names for their intermediate results. Briggs et al. capitalized this fact in semi-pruned SSA form by computing set of names that are live on entry to some basic block in the

program. They called it “non-local” names. Then in the construction of semi-pruned SSA form, ϕ-node is inserted only in iterated dominance frontier for set S that contains only non-local names. Briggs et al. shows that the number of ϕ-nodes in semi-pruned SSA form lie between minimal SSA form and pruned SSA form. The computation of non-local names is cheaper than the liveness analysis. So, semi-pruned form represents a compromise between the time required to perform liveness analysis and the reduction in the number of ϕ-nodes that it allows.

Briggs et al. provides the algorithm for constructing non-local names [5]. That algorithm us used for construction of semi-pruned SSA form in chapter 4.

The renaming process can be improved by efficiently manipulating the stacks of names [5]. The stacks indicate the SSA name of each variable that reach a particular point in the program. The improvement reduces the number of pushes performed in addition to more efficiently locating the stacks that should be popped.

Figure 3.5 shows the three flavors of SSA form that is taken from Briggs et al [5, figure 5]. In minimal SSA form, ϕ-nodes are necessary for all three variables x, y and z. In pruned SSA form, only variable z needs ϕ-node. There is no need of ϕ-nodes for x and y because they are not live. In semi-pruned SSA form, ϕ-nodes for y and z are inserted. ϕ-node for y is still needed because y is live across some block boundary, and that is the limit of the analysis used.

3.3.9 Destruction of SSA form

After translating the program into SSA form and doing optimizations, the compiler must translate SSA form back into an executable form. The compiler must translate the semantics of the ϕ-function into commonly implemented instructions. One simple rule to replace ϕ-node is block b is to insert copy operation into each b’s predecessors. The insertion of copy operation works because ϕ-node maps the incoming values from the predecessors. The copy operation does the same after inserting copy operation at the end

of each b's predecessors. For example consider figure 3.6. Figure 3.6(a) shows the SSA form and figure 3.6(b) shows the code after inserting copy operations.

Cytron et al. describe how to translate out of SSA form by replacing each ϕ-node with some ordinary assignments [3, 4]. Instead of using naïve translation that could yield inefficient object code, they applied two useful optimizations: dead code elimination and storage allocation by coloring to produce efficient object code.

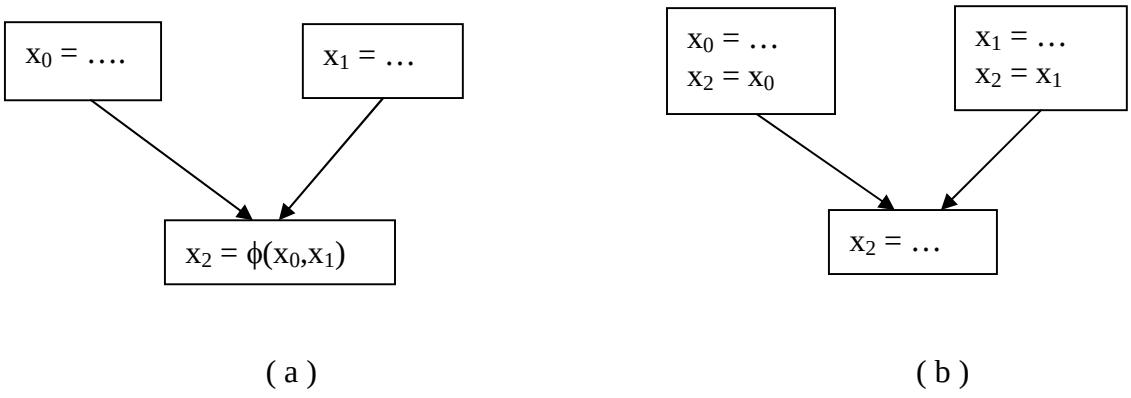

Figure 3.6. (a)SSA form and (b) after insertion of copy operation.

Briggs et al. show how the naïve algorithm that appears to be prior practice can produce incorrect code in cases that involve either "copy folding" or "critical edges". They present an algorithm for replacing ϕ-nodes with copy instructions that generates correct code in the presence of both copy folding and critical edges [5].

3.4 Program Dependence Graph (PDG)

Program Dependence Graph, PDG, is an intermediate representation that contains explicitly both control relationships and data relationships of a program [6]. The PDG for a program consists of a control dependence graph (CDG) and a data dependence graph (DDG). Nodes in a PDG may be basic blocks, statements, individual operators, or constructs at some in-between level. The set of all dependencies for a program may be viewed as inducing a partial ordering on the statements and predicates in the program that

must be followed to preserve the semantics of the original program [6].

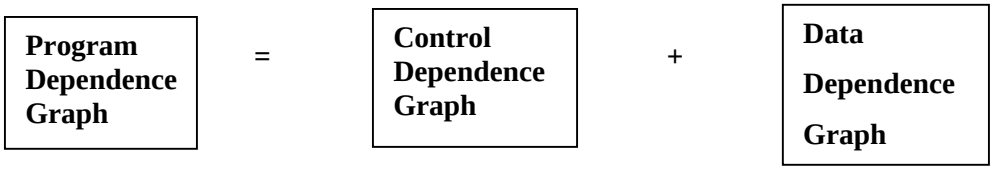

Figure 3.7. Structure of PDG.

Figure 3.7 gives the structure of PDG.

```
S1      a = b + c
S2      if a < 10
S3          d = a * e
S4      print d
```

Figure 3.8. Example of control and data dependencies.

To understand more about dependencies consider the code fragment of fig. 3.8. If S3 execute before S1, then the value computed by S3 would be incorrect because S3 needs value of a that is computed in S1. That is, dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. This type of dependency is called *data dependences*. Now consider S2 and S3. Execution of statement S3 depends on the statement S2. This type of dependency is called *control dependences.*

The following two subsections will describe about the construction of CDG and DDG.

3.4.1 Control Dependence Graph (CDG)

In this subsection, first few important definitions are presented that are needed for the construction of CDG.

A *control flow graph* is a directed graph G augmented with a unique *entry* node ENTRY and a unique *exit* node EXIT such that each node in the graph has at most two successors.

Assume that nodes with two successors have attributes “T” (true) and “F” (false) associated with the outgoing edges in the usual way. Further assume that for any node N in G there exists a path from ENTRY to N a path from N to EXIT [3, 6].

A node M is *post-dominated* by a node N in G if every directed path from M to EXIT (not including M) contains N [6].

Let M and N are the nodes of a control flow graph G. Then node N is *control-dependent* on node M if and only if

1) there exists a directed path P from M to N with any Q in P (excluding M and N) post-dominated by N and
2) M is not post-dominated by N [6].

The construction of CDG consists two steps. In first step, we construct the basic CDG and in the second step so-called *region nodes* are added to it. To construct the basic CDG, special predicate node START is added to the control flow graph with its “T” edge running to the ENTRY node and its “F” edge to EXIT. This resulting graph is G^+. Next step is to construct the post-dominance relation on G^+, which can be displayed as a tree. Computing post-dominators in the control flow graph is equivalent to computing dominators in the reverse control flow graph [3]. The computation of control dependences from the dominators in the reverse control flow graph will be discussed soon. After obtaining the dominator tree, control dependencies are determined by examining certain control flow graph edges and annotating nodes on corresponding tree paths. Let S be the set of edges (M, N) in G^+ such that N does not post-dominate M. Now the control dependence determination algorithm proceeds by examining each edge (M, N) in S. Let L be the least common ancestor of M and N in the post-dominator tree. Ferrante et al. show that either L is M of L is the parent of M in the post-dominator tree [6]. So consider the two cases for L. In the case where L is parent of M, all nodes in the post-dominator tree on the path from L to N, including N but not L, should be made control dependent of M. In the case where L is equal to M, all nodes in the post-dominator tree on the path from M to N, including M and N, should be made control dependent on M. To achieve this, traverse backwards from N in the post-dominator tree until M’s parent (if

it exists) is reached, marking all nodes visited before M's parent as control dependent on M. Ferrante et al. provide the correctness of this construction [6].

Consider an augmented control flow graph in fig. 3.9 (a). Its corresponding post-dominator tree is figure 3.9 (b). The basic CDG is shown in fig. 3.10 (a) after obtaining set S, which contains (START, ENTRY), (1, 2), (1, 3) and (4, 6).

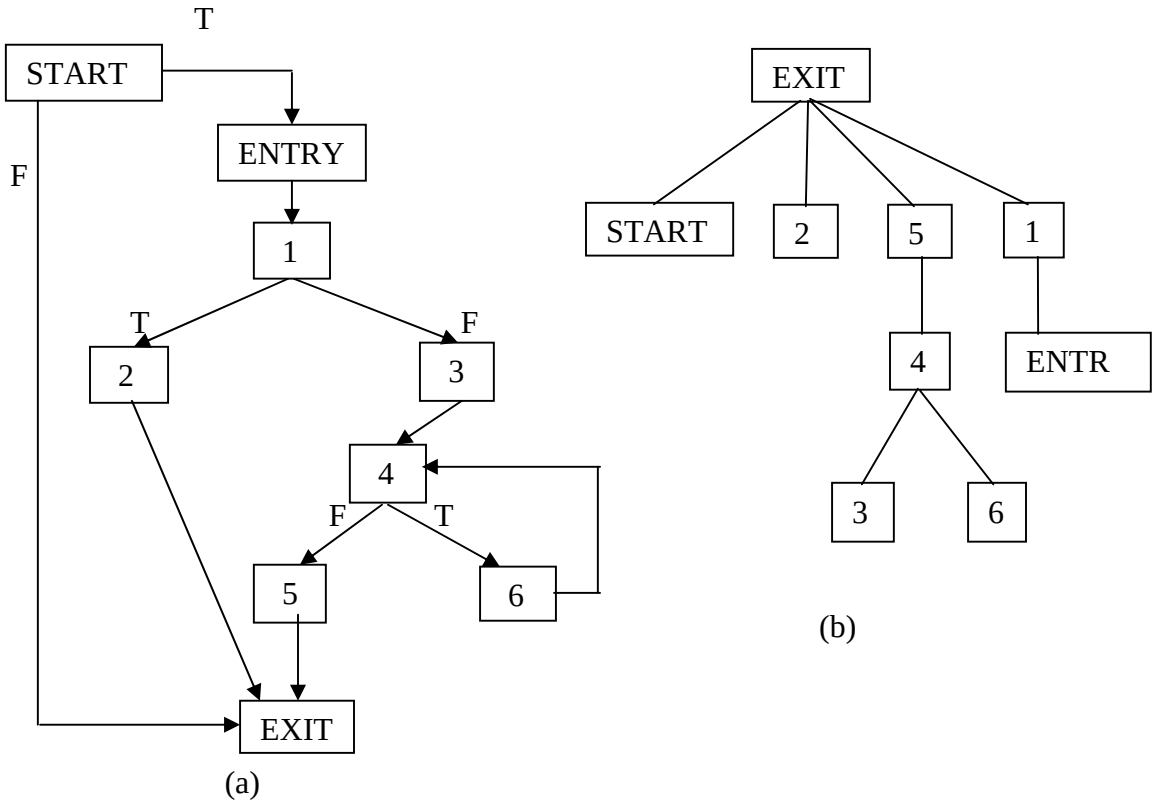

Figure 3.9. (a) Augmented control flow graph and (b) its post-dominator tree.

The final step of the CDG construction is the addition of region nodes to summarize the set of control conditions for a node and group all nodes with the same set of control conditions together. See fig. 3.10 (b). Region nodes are also inserted so that predicate nodes will have only two successors, as in the control flow graph. Ferrante et al. show how region nodes can be inserted in CFG by extending the concept that any nodes having a proper containment of their sets of control dependences must be adjacent to one another on some path in the post-dominator tree [6].

CDG can be efficiently constructed efficiently using the method provided by Cytron et al. [3]. The *reverse control flow graph* RCFG has the same nodes as the control flow graph CFG, but has an edge Y→X for each edge X→Y in CFG. The roles of ENTRY and EXIT are also reversed. Cytron el at. gave the proof that the post-dominator relation on CFG is the dominator relation on RCFG. They provide an algorithm for computing control dependences in **O**(E + size(RDF)), where RDF is the dominance frontier in RCFG [3].

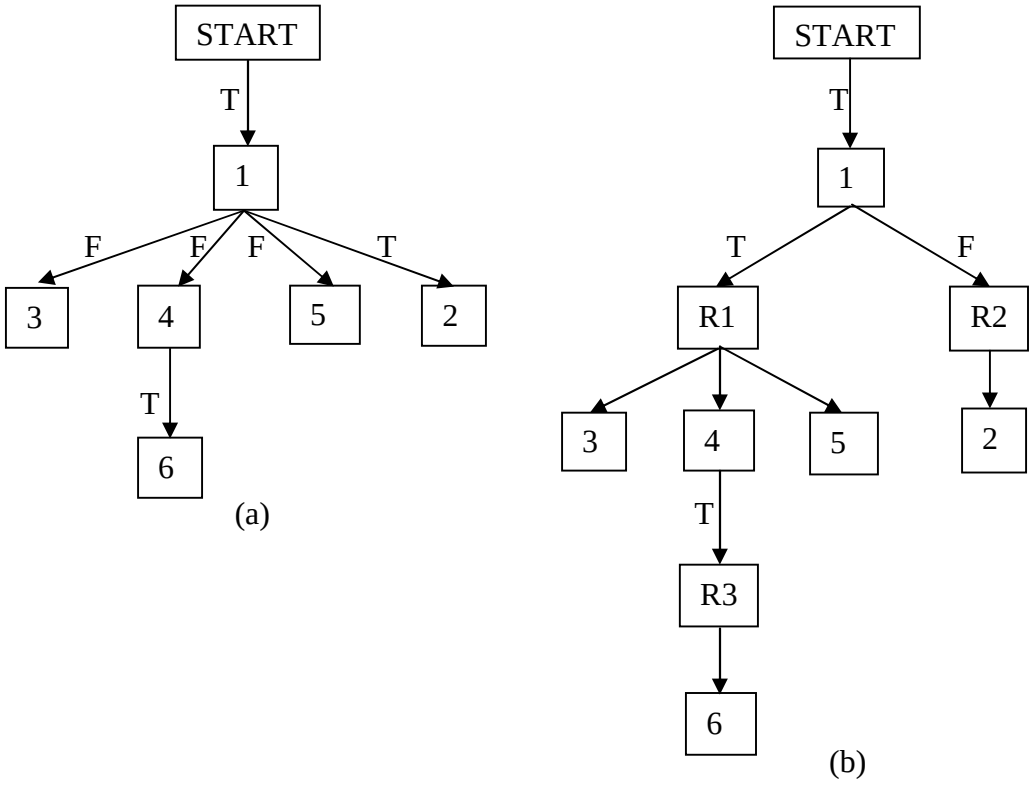

Figure 3.10. (a) basic control dependence graph and (b) CDG with region nodes.

3.4.2 Data Dependence Graph (DDG)

The Data Dependence Graph, DDG, represents the data dependence edges in the PDG. If there is a data dependence edge between two basic blocks (A, B), then there is definition in basic block A and that definition is used in basic block B. That is, explicit def-use chains [1].

Other edges are also necessary for certain transformations. Ferrante el at. described

incremental data dependence update algorithm that requires *output dependence* edges [6]. Let S1 and S2 are two statements in the program and S1 comes before S2. If both statements set the value of some variable, then it is called an output dependence. If S1 uses some variable's value, and S2 sets it, then it is called an *antidependence* between them. If both statements read the value of some variable, there is an *input dependence between* them. And finally, if S1 sets a value that the latter uses, then there is *true dependence* or *flow dependence* [2].

```
S1      i = 0
S2      sum = 0
S3      while i != 100
S4          sum = sum + 1
S5          i = i + 1
S6      print I
```

Figure 3.11. Example of data dependences.

Consider a fragment of code in fig. 3.11. There is true dependence between S1 and S3, S1 and S5, since S1 sets value of i that are used in S3 and S5. An antidependence can be found in S3 and S5 because S3 uses the value of i and S5 sets the value of i. S1 and S5 have output dependence, since both set the value of i. Similarly, S2 and S4 also have output dependence because both define the value of sum. Statements S3 and S6 have input dependence because both read value of i. And finally, S3 and S5 also have input dependence because both read value of sum.

The main property of PDG is the exposing of potential parallelism. That property can not be found in the control flow graph because CFG contains unnecessary sequencing between operations. Since dependences in the PDG connect computationally relevant parts of the program, many code improving transformations, like vectorization and code motion [6], require less time to perform than with other program representations. Ferrante el at. show how PDG can be applied in applications like: detection of parallelism, node splitting, code motion, loop fusion and software development environment [6].

CHAPTER 4

4 IMPLEMENTATION

In this chapter, steps for the implementation have been provided for the selected intermediate representations, Static Single Assignment (SSA) [3] and Program Dependence Graph (PDG) [6]. SUIF2 [31] and Machine SUIF [32] systems are used for the implementation SSA and PDG.

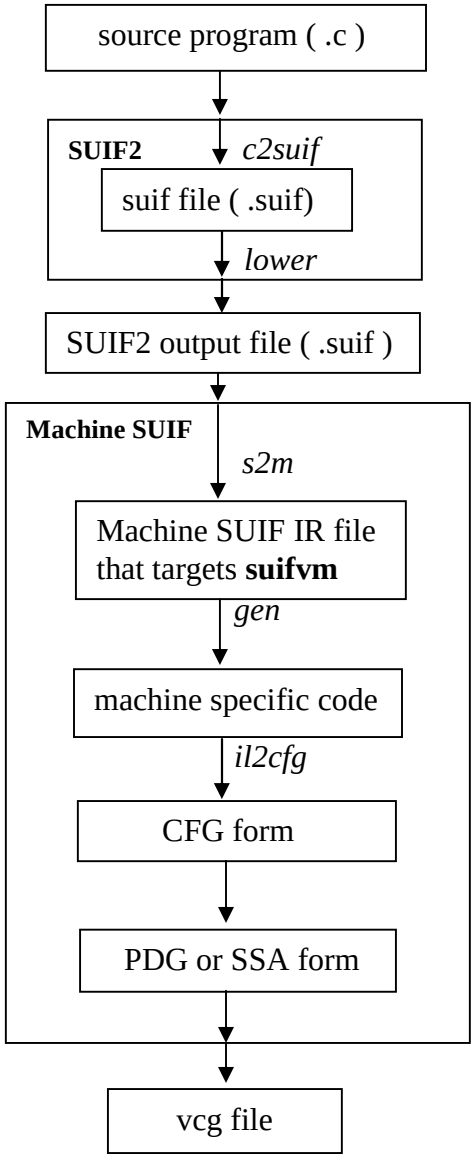

Figure 4.1. Implementation Structure.

Section 4.1 presents the actual algorithm for implementation of three flavors SSA form. Section 4.2 presents algorithm for implementing PDG. Section 4.3 shows how the passes are used in Machine SUIF.

Figure 4.1 shows the broad view of our implementation steps. Appropriate benchmark programs have been taken for this dissertation. These input programs are converted into suif file using pass *c2suif*. Next, SUIF transformations are applied that make IR file acceptable for lowering into Machine SUIF using *lower* pass. Complex operations in the SUIF2 IR are decomposed into simpler operations; for example, **if** statements and **for** loops are dismantled into branch and jump operations. After that *s2m* pass is used to convert IR into SUIFvm, the architecture independent assembly language of Machine SUIF (*do_s2m* in the shell). The *gen* pass does most of the machine-specific code generation. This pass takes parameter for specific target machine, *–target_lib x86* for x86 architecture. The *il2cfg* pass transforms the IR from simple instruction-list form to CFG form. Finally, the developed passes are applied which convert CFG form into SSA form or PDG form according to the pass. The implemented IR can be viewed in X11 using VCG [33] tool after writing IR in VCG format that Machine SUIF provide.

4.1 Implementing Static Single Assignment (SSA)

SSA is already implemented in Machine SUIF. But in this section algorithms are presented that are used by Machine SUIF to implement SSA in detail. The Machine SUIF static single-assignment library translates an optimization unit into or out of SSA form [3]. Three variants of SSA form can be implemented very easily. The implementation follows the design described by Briggs et al. at Rice University [5]. The following subsections show three flavors of SSA form have been implemented in Machine SUIF.

4.1.1 Implementing minimal SSA form

The Machine SUIF SSA Library implements the algorithm for constructing minimal SSA form by algorithm shown in fig.4.2. As stated in section 3.3 of chapter 3, there consists of

two steps. The first step is determining locations for ϕ-nodes and second step is the renaming of variables.

```
/* STEP 1: Determine locations for ϕ-nodes */
Calculate the dominator tree and dominance frontiers
For each variable, V
        S ← { blocks containing an assignment to V }
        Place a ϕ-node for V in the iterated dominance frontier of S.

/* STEP 2: Rename each variable, replace V, with the appropriate Vi */
For each variable, V
        Counters[V] ← 0
        Stacks[V] ← emptystack()
SEARCH(start)
/* Recursively walk the dominator tree, renaming variables */
SEARCH(block)
        For each ϕ-node, V ← ϕ(…), in block
                i ← Counters[V]
                Replace V by Vi
                Push(i, Stacks[V])
                Counters[V] ← i +1
        For each instruction, V ← x op y, in block
                Replace x with xi , where i ← top(Stacks[x])
                Replace y with yi , where i = top(Stacks[y])
                i ← Counters[V]
                Replace V by Vi
                Push i onto Stacks[V]
                Counters[V] ← i + 1
        For each successor, s, of block
                j ← whichPred(s, block)
```

continue…

```
                                For each ϕ-node, p, in s
                                    V ← j^th operand of p
                                    Replace V with V_i, where i ←top(Stacks[V])
                            For each child, c, of block in the dominator tree
                                SEARCH(c)
                        For each instruction, V ← x op y, or ϕ-node, V ← ϕ(…), in block
                                Pop(Stacks[V])
```

Figure 4.2. Algorithm for building minimal SSA form.

The first step of placing the ϕ-node is the calculation of the dominator tree for the CFG and calculation of dominance frontiers for the nodes in the CFG. More about the dominator tree can be found in section 3.3.3 of chapter 3. Finding the dominance frontiers, DF, for the nodes using the definition is not efficient, so Cytron et al. provides an efficient way of finding dominance frontiers of the nodes by calculating DF_{local} and DF_{up} [3].

$$DF(X) = DF_{local}(X) \cup \Ć_{Z \in Children(X)} DF_{up}(Z)$$

More about DF_{local} and DF_{up} can be found in section 3.3.4. Cytron et al. give an algorithm for finding dominance frontiers which runs in $\mathbf{O}(E + N^2)$. Figure 4.3 presents that algorithm. At first, DF_{local} is calculated node X and then DF_{up} is calculated for each children of X as shown in fig. 4.3.

Now, after the calculation of dominance frontiers of each CFG nodes, iterated dominance frontier are calculated for each CFG nodes, denoted by $DF^+(X)$. Cytron et al. show that ϕ-nodes for V are required only in blocks in $DF^+(S)$, where S contains the set of nodes of CFG that has assignments for variable V [3]. Other efficient algorithms for placing ϕ-

nodes are also available [8,9], but team at Harvard University used algorithm provided by Cytron et al. [3] for Machine SUIF.

```
For each X in a bottom-up traversal of the dominator tree
    DF(X) = Φ
    For each successor, Y, of  X in the CFG
        /* local */
        if idom(Y) ≠ X then
            DF(X) = DF(X) ∪ {Y}
    For each child, Z, of X in the dominator tree
        For each Y ∈ DF(Z)
            /* up */
            If idom(Y) ≠ X then
                DF(X) = DF(X) ∪ {Y}
```

Figure 4.3. Algorithm for calculating DF(X) for each CFG node X.

Figure 4.4 shows the algorithm for placing ϕ-nodes in the minimal SSA form [3]. The main outer loop of this algorithm is performed once for each variable V. Cytron el at. called this algorithm worklist algorithm because it uses worklist, W, data structure for representing iterated dominance frontier [3]. There are few data structures used in this algorithm. W is the worklist of CFG nodes being processed. In each iteration of this algorithm, W is initialized to the set of nodes S that contain assignments to V. Each node X in the worklist ensures that each node Y in DF(X) receives a ϕ-node. And each iteration terminates when the worklist becomes empty. Work(*) is an array of flags, one for each node. Work(X) indicates whether X has ever been added to W during the current iteration of the outer loop. HasAlready(*) is an array of flags, one flag for each node, where HasAlready(X) indicates whether a ϕ-node for V has already been inserted at X. The flags Work(X) and HasAlready(X) are independent. These two flags are necessary because the property of assigning to V is independent of the property of needing a ϕ-node for V.

```
IterCount ← 0
For each node X
    HasAlready(X) ← 0
    Work(X) ← 0
W ← Φ
For each variable V
    IterCount ← IterCount + 1
    For each X ∈ S
        Work(X) ← IterCount
        W ← W ∪ {X}
    While W ≠ Φ
        Take X from W
        For each Y ∈ DF(X)
            If HasAlready(Y) < IterCount then
                Place ⟨ V ← ϕ(V,…,V) ⟩ at Y
                HasAlready(Y) ← IterCount
                If Work(Y) < IterCount then
                    Work(Y) ← IterCount
                    W = W ∪ {Y}
```

Figure 4.4 Algorithm for placing ϕ-node.

The last step of constructing minimal SSA form is to rename variables to create the single assignment property. This is accomplished in a single recursive walk of the dominator tree, shown in the procedure SEARCH in fig. 4.2. The procedure SEARCH maintains two data structures for each name in the original code. The first one is Counters[V] which contains the subscript that will be assigned to the next definition of V. And the second one, Stacks[V], holds the current subscript for V. For each new definition of V, SEARCH renames V with the subscript from Counters[V], pushes that value onto Stacks[V], and

increments Counters[V]. Now in the first step of renaming, it rewrites variable names, incrementing the various counters and pushing new names onto the appropriate stacks. And in the next step of renaming, it rewrites ϕ-node parameters in any successor blocks in the CFG so that the name inherited from the current block has the current subscript. To rename the parameters of ϕ-node, it uses the whichPred function to determine which ϕ-node parameter in the successor corresponds to the current block. To continue the search, it recurses on each child in the dominator tree. After the recursion is complete, it processes the current block again, to pop from each stack any subscripts added while processing the block.

4.1.2 Implementing pruned SSA form

To construct pruned SSA form, liveness analysis is needed first to insert ϕ-nodes. The liveness analysis analyses the variables in that are live on entry to the block [1, 2]. Liveness analysis is a backward data flow problem. Section 3.4 of chapter 3 describes more about pruned SSA form. Machine SUIF contains liveness analyzer. The Machine SUIF Bit-Vector Data-Flow-Analysis (BVD) Library provides liveness analyzer [32].

The algorithm for constructing pruned SSA form is similar to the construction of minimal SSA form. In figure 4.2, live information has to compute and modify the first step where ϕ-nodes are inserted. The pruned SSA form construction inserts ϕ-node for V in every node $N \in DF^{+}(S)$, where $V \in$ Liveness(N) whearas algorithm presented in fig. 4.2 inserts ϕ-node for V in every node $N \in DF^{+}(S)$. The renaming process is same as the minimal SSA form.

4.1.3 Implementing semi-pruned SSA form

The construction of semi-pruned SSA form is based on the fact that many of the names in a program being converted to SSA form are local to a single basic block [5]. They are compiler-generated temporaries that are never live across a control flow edge. Therefore,

the set of names that are live on entry to some basic block in the program, called “non-local” names, are computed. The construction only computes for the set S that contains assignments of non-local names. Section 3.5 of chapter 3 provides more about semi-pruned SSA form.

In construction of semi-pruned SSA form, non-local names should be discovered. Briggs el at. provide an algorithm, shown in fig. 4.5, to discover the non-local names [5]. Machine SUIF used that algorithm. This algorithm requires only two sets, non-locals and killed. The algorithm makes a simple forward pass over each basic block. If an operand that has not already been defined within the block, i.e. not in the killed set, then that operand must be non-local names. This algorithm is very simple than the algorithm for pruned SSA form. Now after finding the non-local names, ϕ-node is inserted for every non-local name, V, by computing iterated dominance frontier of set S, where the set S contains assignments to V. Finally, renaming of each variable in semi-pruned SSA form is same as renaming of each variable in minimal SSA form. The renaming process can be found in second part of the algorithm presented in fig. 4.2.

```
non-locals ← Φ
For each block B
    killed ← Φ
    For each instruction i ← x op y in B
        If x ∉ killed  then
            non-locals ← non-locals ∪ { x }
        If y ∉ killed  then
            non-locals ← non-locals ∪ { y }
        killed ← killed ∪ { i }
```

Figure 4.5. Algorithm for finding non-local names.

4.2 Implementing Program Dependence Graph (PDG)

The construction of Program Dependence Graph (PDG) can be done in two parts. In the first part, Control Dependence Graph (CDG) is constructed. And in the second part, Data Dependence Graph (DDG) is constructed.

4.2.1 Implementing Control Dependence Graph (CDG)

Control dependence information can be calculated using the algorithm provided by either Ferrante el at. [6] or Cytron el at. [3]. This dissertation has used the algorithm that is provided by Cytron el at., shown in fig. 4.6. The first step of this algorithm is to build reverse control flow graph RCFG and dominator tree for RCFG. Next dominance frontier RDF for RCFG is calculated using the algorithm presented in fig. 4.3. Luckily Machine SUIF contains the calculation of RDF in the Machine SUIF Control Flow Analysis (CFA) Library [21]. The **DominanceInfo** class computes dominator sets, the dominator tree, and dominance frontiers in either the forward or reverse graphs. This study has used the method **find_reverse_dom_frontier( )** to calculate RDF for RCFG. After calculating dominance frontier for each node in RCFG, control dependence set CD is initialized for each node with null. Now finally, CD is computed for each node shown in fig. 4.6.

Build RCFG
Build dominator tree for RCFG
Apply the algorithm for finding dominance frontier RDF for RCFG

For each node X do $CD(X) \leftarrow \phi$
For each node Y
    For each $X \in RDF(Y)$
        $CD(X) \leftarrow CD(X) \cup \{ Y \}$

Figure 4.6. Algorithm for computing the set CD(X) nodes that are control dependent on X.

After computing the set CD for each node, CDG was built according to the information found in the set CD. CDG is built in Machine SUIF using appropriate interfaces that are present in the Machine SUIF Control Flow Graph Library [20]. Region nodes are not inserted in the CDG for simplicity. The purpose of region nodes is to group together all the nodes that have the same control dependence on a particular predicate node, giving each predicate node at most two successors.

4.2.2 Implementing Data Dependence Graph (DDG)

In this work, Data Dependence Graph (DDG) is constructed using the def-use chains [1, 2]. The Machine SUIF Bit-Vector Data-Flow-Analysis (BVD) Library provides the def-use analyzer for each instruction in basic block [14]. The **DefUseAnalyzer** class of this library is used. Definition set and use set are taken from this class using methods **defs_set( )** and **uses_set( )** respectively. Each time two nodes X and Y of CFG are taken and investigated if there is any data dependence edge. Data dependence is investigated between X and Y in two steps. Firstly, if there is data dependence from X to Y, data dependence edge (X,Y) is inserted. That is, if Y use the operand that is defined at X and there is no redefinition between X and Y of that operand then data dependence edge (X,Y) is inserted. Secondly, if there is data dependence from Y to X, data dependence edge (Y,X) is inserted in PDG.

4.3 Inputs and Outputs

In this section, tools that are used, input programs, and outputs are presented. A sample input C program is presented in fig. 4.7. Following passes of Machine SUIF are used to convert C program into its corresponding CFG form:

c2suif sample.c
do_lower sample.suif sample.lsf
do_s2m sample.lsf sample.svm
do_gen –target_lib x86 sample.svm sample.xil
do_il2cfg sample.xil sample.cfg

```
/* sample.c */
int main ( )
{
    int i, sum = 0;
    i = 1;
    while ( i < 100 )
    {
        sum += i;
        i++;
     }
   printf("The sum of first 100 natural numbers is %d\n", sum);
   return 0;
 }/* end of main */
```

Figure 4.7. Sample input C program sample.c.

do_fg2ssa pass is used to convert cfg formed input program into SSA form. This pass writes the SSA into VCG format, *ssa.vcg*. Figure 4.8 shows the SSA graph of our sample program *sample.c*.

After that, *do_cfg2pdg* pass is used to convert cfg formed input program into PDG. This pass also writes the PDG into VCG format called *pdg.vcg* so that the actual PDG can be seen in X11. Figure 4.9 presents PDG of our sample program in X11.

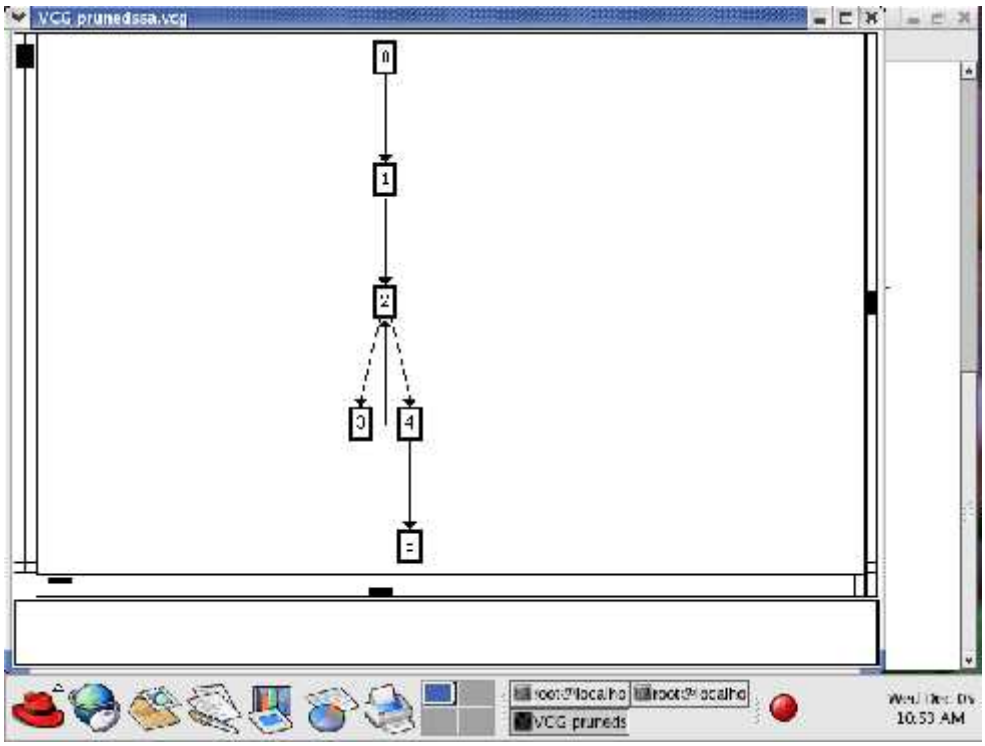

Figure 4.8 Visualization of SSA form in X11.

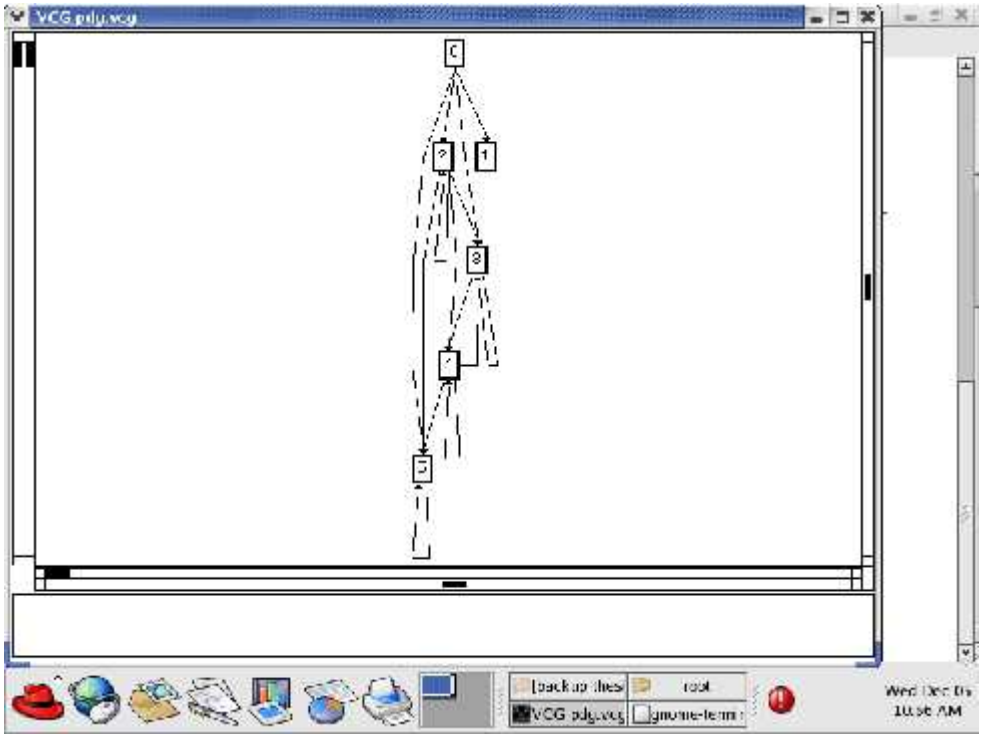

Figure 4.9 Visualization of PDG in X11.

CHAPTER 5

5 TESTING AND ANALYSIS

In this chapter,the various results that are obtained in experiments are presented. Selected IRs, Program Dependence Graph (PDG) [6] and Static Single Assignment (SSA) [3], have been tested in GNU/Linux platform in 3.00 GHz. Intel Pentium 4 with 1 GB RAM. Section 5.1 presents the empirical comparison and analysis between selected IRs.

5.1 Empirical Comparison

Selected IRs, PDG and SSA, are implemented in C++ using Machine SUIF [32]. In particular, this dissertation hasn't implemented SSA, Machine SUIF provides simple interface to create SSA using the Machine SUIF Static Single Assignment Library [13], as described in chapter 4. Both PDG and SSA operate on CFG.

The time taken to construct PDG and three flavors of SSA form are measured, shown in table 5.1. The construction time presented in table 5.1 is the mean of twenty tests. Also, number of ϕ-nodes are compared in three flavors of SSA form. Various benchmark input programs (C) are taken. Input programs are taken carefully to cover as many areas as possible. For example, data-structure, sorting etc. Three graphs from table 5.1 are presented in section 5.2 for analysis to give the clear view. In first graph, the comparison between construction time of PDG and three flavors of SSA form is shown, fig. 5.1. In second graph, comparison of number of edges between PDG and three flavors of SSA form is presented, shown in fig. 5.2. In the last graph, comparison of number of ϕ-nodes between three flavors of SSA form is presented, shown in fig. 5.3.

Program	Program Dependence Graph (PDG)		Static Single Assignment (SSA) form						
			Edge	Minimal SSA		Pruned SSA		Semi-pruned SSA	
	Time (m.s.)	Edge		Time (m.s.)	¤-nodes	Time (m.s.)	¤-nodes	Time (m.s.)	¤-nodes
guessinggames	311.5	149	31	6.5	110	5.5	0	4.0	1
linklist	302.0	189	30	8.5	153	8.0	9	6.5	12
bubblesort	264.0	192	33	9.5	199	6.5	5	5.0	6
matrix	203.5	92	22	10.0	282	6.0	5	7.0	9
shellsort	169.0	174	38	10.0	194	6.5	8	4.5	15
primefactorSOE	135.0	145	32	9.0	192	5.5	7	4.5	9
insertionsort	122.5	108	27	10.0	135	6.5	6	6.0	9
selectionsort	101.5	85	21	6.5	97	5.5	6	4.0	8
euclid	32.0	15	6	4.0	18	5.0	8	3.5	8

Table 5.1. Comparing construction time, edges between PDG and SSA, and number of ϕ-nodes between three flavors of SSA form.

5.2 Analysis

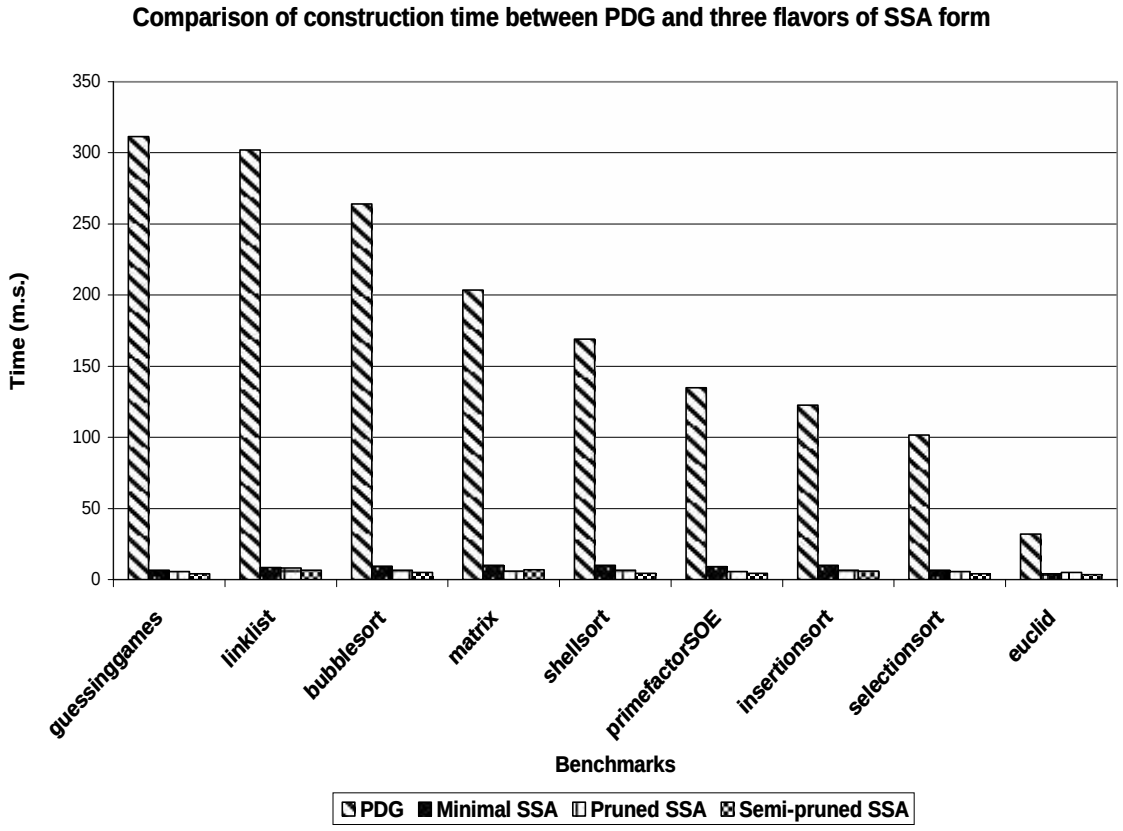

Figure 5.1. Comparison of construction time between PDG and three flavors of SSA form.

From graph in fig. 5.1, x-axis represents benchmark programs and y-axis represents construction time in milliseconds. The construction time of PDG is expensive than any of the SSA forms. In average, SSA form took 96% less time to construct than PDG. This is because PDG contains two sub-graphs explicitly, one is Control Dependence Graph (CDG) and another is Data Dependence Graph (DDG). SSA contains single graph and its control structure is same as CFG. So constructing PDG takes much time than SSA.

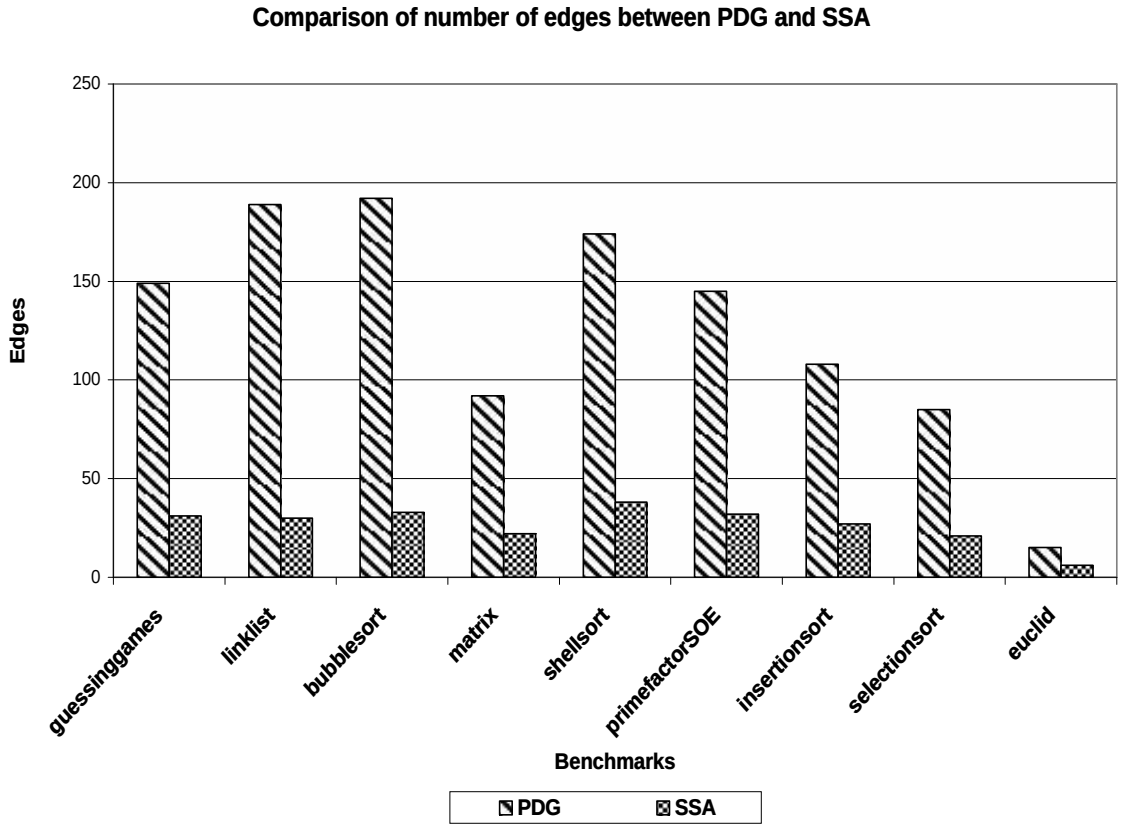

Figure 5.2. Comparison of number of edges between PDG and three flavors of SSA form.

The work found that the number of edges in PDG is higher than SSA form, shown in fig. 5.2, where x-axis represents benchmarks and y-axis represents number of edges. In average, SSA form contained 79% less edges than PDG. PDG contains two sub-graphs, CDG and DDG. And SSA contains edges exactly from CFG. So PDG contains control dependence edges and data dependence edges compare to only control dependence edges of SSA form.

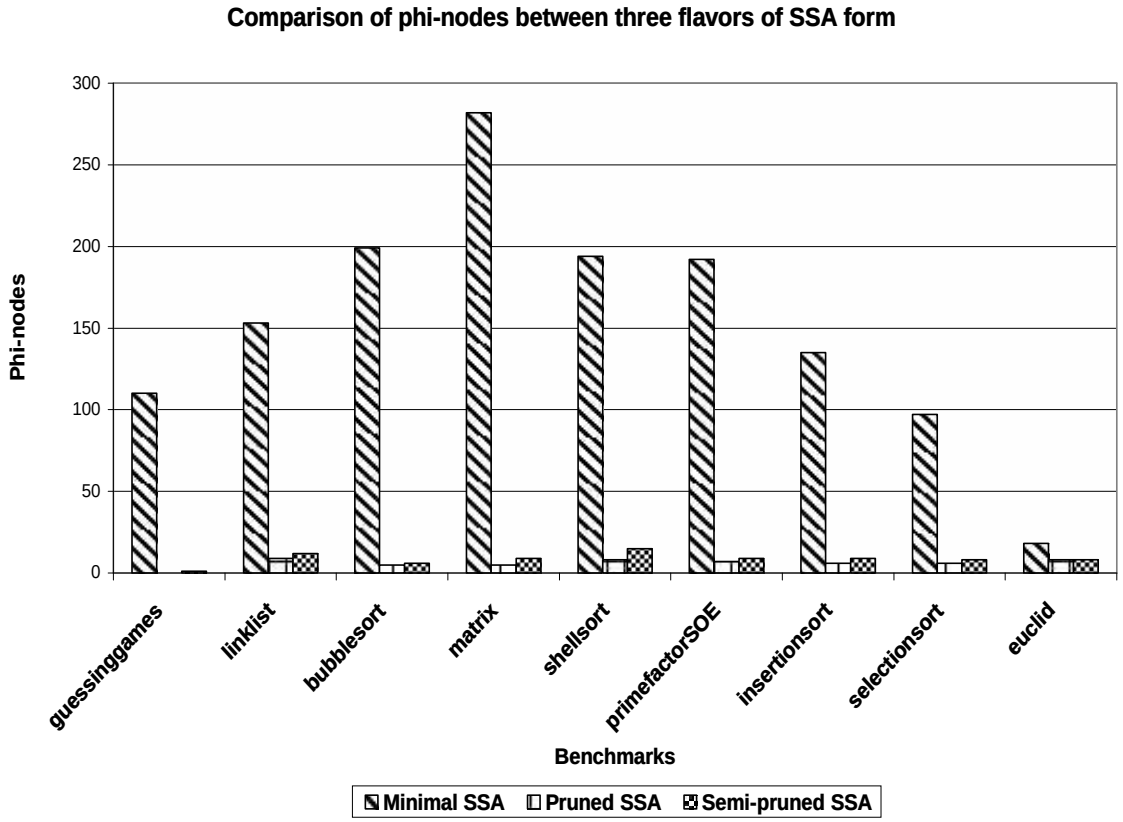

Figure 5.3. Comparison of number of ϕ-nodes between three flavors of SSA form.

Finally, this study found that the number of ϕ-nodes in minimal SSA form is higher than in pruned SSA form and semi-pruned SSA form, shown in fig. 5.3. In this figure x-axis represents benchmark programs and y-axis represents number of ϕ-nodes. Semi-pruned SSA form contains much less ϕ-nodes than minimal SSA form but slightly more ϕ-nodes than pruned SSA form. In our observations, pruned SSA form contained 96% less ϕ-nodes than minimal SSA form and only 30% less ϕ-nodes than semi-pruned SSA form. Semi-pruned SSA form contained 94% less ϕ-nodes than minimal SSA form. But more importantly semi-pruned SSA form took 18% less time to construct than pruned SSA form and 39% less time to construct than minimal SSA form. Pruned SSA form took 26% less time to construct than minimal SSA form. This is because minimal SSA form inserts ϕ-nodes without doing any analyses. But pruned SSA form inserts ϕ-nodes after liveness analysis and doesn't contain dead ϕ-nodes. Semi-pruned SSA form inserts ϕ-nodes only

if variable is live on some basic block entry. Semi-pruned looks promising because its construction time is low and ϕ-nodes are lesser than other two forms.

Even though PDG have more edges and expensive to construct, PDG have few major advantages over other IRs. For example, PDG can exploit potential parallelism and very useful in multiprocessor system where parallel execution of instruction could take place. PDG could be very useful in optimizations like program slicing, node splitting, code motion, and loop fusion [6].

CHAPTER 6

6 CONCLUSIONS

6.1 Summary

This dissertation has worked on the problem of selecting right intermediate representation (IR) for the compiler and provided comparative analyses of various IRs.

In the past, simple IR for compiler is sufficient for analyses and optimizations. But as the development of complier technology occurred and new analyses and optimizations are needed, many IRs were proposed. Compiler writer found very difficulty in selecting right IR for their compiler. This study has provided comparative analyses of different IRs in terms of construction time, number of edges etc. and gave efficient IR with reasons.

In the earlier part of this dissertation, many important IRs and their taxonomies are presented. Motivation, tools needed and used for this dissertation are also presented.

Later, two important IRs, SSA and PDG, are selected and analyzed in greater detail. This dissertation has preseted how dominance information of CFG could be used for constructing three flavors of SSA form according to Cytron el at. [3]. CDG is constructed that is presented by Cytron el at. [3] for PDG. DDG is constructed using def-use information.

Finally, implementation steps of selected IRs using tools are presented. After testing with various benchmark programs, this study has found that PDG is expensive in terms of construction time and number of edges with any of the three flavors of SSA form. In particular, SSA form took 96% less time to construct and contained 79% less edges than PDG. The study found that pruned SSA contained 96% less ϕ-nodes than minimal and 30% less ϕ-nodes than semi-pruned. Semi-pruned contained 94% less ϕ-nodes than minimal. To construct semi-pruned it took 18% less time than pruned and 39% less than minimal. Pruned took 26% less time to construct than minimal. But this work has also

focused on the fact that PDG is important if multiprocessor system is used and useful for optimizations like program slicing, node splitting, code motion, and loop fusion. The comparative study presented in this work is useful for compiler designer.

6.2 Future work

This dissertation has provided comparative analyses for selecting appropriate IR for compiler that gave compiler writer a huge benefit of selecting IR and could save much of the time in design and implementation of compiler.

This study has selected PDG and SSA for comparison. In the implementation of PDG, insertion of region nodes was avoided for simplicity. In the future, insertion of region nodes are needed. Efficient construction of DDG in the PDG and compare number of nodes between PDG and any other graph IRs will be acounted. Other important IRs are needed for comparison in the future. Moreover, new optimization passes have to be developed and empirical comparison according to those optimizations need to be presented. This dissertation has used x86 as target architecture, other important architectures will be used as target architecture in the next work.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. Compilers: Principles, Techniques, and Tools.Addison-Wesley, 1986.

[2] Muchnick, S. S. Advanced Compiler Design and Implementation. Morgan Kauffman, 1997.

[3] Cytron, R., Ferrante, J., Barry K. Rosen, Wegman, M. N. and Zadeck, F., K. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, 13(4):451–490, Oct 1991.

[4] Cytron, R., Ferrante, J., Barry K. Rosen, Wegman, M. N. and Zadeck, F., K. An Efficiently Computing Static Single Assignment Form. In Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pages 25-35, Austin, Texas, January 1989.

[5] Briggs, P., Harvey, T., and Simpson, L. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. Software Practice and Experience, 28(8), pp.859-881, July 1998.

[6] Ferrante, J., Ottenstein, K., and Warren, J. The Program Dependence Graph and its Use in Optimization. ACM Transactions on Programming Languages and Systems, 9(3):319-349, July 1997.

[7] Briggs, P., Harvey, T., and Simpson, L. Static Single Assignment Construction. Implementation document, 1996.

[8] Cytron, R. and Ferrante, J. Efficiently computing □-nodes on-the-fly. ACM Transactions on Programming Languages and Systems, 17(3):487-506, May 1995.

[9] Sreedhar, V. C., and Gao, G. T. A linear time algorithm for placing □-nodes. In Conference Record of POPL ’95: 22jn ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 62-73, San Francisco, California, January 1995.

[10] Briggs, P., and Cooper, K. D., and Torczon, L. Improvements to graph coloring register allocation. ACM Transactions on Programming Languages and Systems, 16(3):428-455, May 1994.

[11] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. Register allocation via coloring. Computer Languages, 6:47-57, January 1981.

[12] Lengauer, T., and Tarjan, R. E. A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems, 1(1):121-141, July 1979.

[13] Holloway, G. The Machine-SUIF Static Single Assignment Library. Harvard University, July 15 2002.

[14] Holloway, G., and Smith, M. D. The Machine-SUIF Bit-Vector Data-Flow Library. The Machine-SUIF documentation set, Harvard University, July 15 2002.

[15] Alpern, B., Wegman, M., and Zadek, F. Detecting Equality of Variables in Programs.InConference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages, 1988.

[16] Rosen, B. K., Wegman, M. N., and Zadek, F. K. Global Value Numbers and

Redundant Computations. Conf. Rec. Fifteenth ACM Symp. On Principles of Programming Languages, January 1988.

[17] Wegman, M. and Zadek, F. Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems, 13920: 181-210, April 1991.

[18] Singer, J. Static Program Analysis based on Virtual Register Renaming. Ph.D. thesis, Chirst's College, March 2005.

[19] Allen, F. E. Control Flow Analysis. Sigplan Notices, July 1970.

[20] Holloway, G., and Smith, M. D. The Machine-SUIF Control Flow Graph Library. The Machine-SUIF documentation set, Harvard University, July 15 2002.

[21] Holloway, G., and Smith, M. D. The Machine-SUIF Control Flow Analysis Library. The Machine-SUIF documentation set, Harvard University, July 15 2002.

[22] Ananian, C. S. The Static Single Information Form. Ph.D. thesis Princeton University, 1997.

[23] Click, C., Paleczny, M. A Simple Graph-based Intermediate Representation. In Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, pages 35–49, 1995.

[24] Weise, D., Crew, R., Ernst, M., and Steensgaard, B. Value Dependence Graphs: Representation without Taxation. In Conference Record of the Twenty-first ACM Symposium on the Principles of Programming Languages, 1994.

[25] Pingali, R., Beck, M., Johnson, R., Moudgill, M., and Stodghill, P. Dependence Flow Graphs: An Algebraic Approach to Program Deendences. In Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 67–78, 1991.

[26] Balance, R. A., Maccabe, A. B., Ottenstein, K. J. Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In Proceedings of the SIGPLAN 90 Symposium on Compiler Construction. SIGPLAN Not. (ACM) 25, 6 (June 1990), 257-271.

[27] Holloway, G., and Smith, M. D. An Extender's Guide to the Optimization Programming Interface and Target Descriptions. The Machine-SUIF documentation set, Harvard University, July 15 2002.

[28] Holloway, G., and Smith, M. D. The Machine-SUIF Cookbook. The Machine-SUIF documentation set, Harvard University, July 15 2002.

[29] Holloway, G., and Smith, M. D. The Machine-SUIF Machine Library. The Machine-SUIF documentation set, Harvard University, July 15 2002.

[30] Holloway, G., and Smith, M. D. A User's Guide to the Optimization Programming Interface. The Machine-SUIF documentation set, Harvard University, July 16 2002.

[31] SUIF. Stanford University Intermediate Format.

http://www.suif.stanford.edu

[32] Machine SUIF. Harvard University, USA.
http://www.eecs.harvard.edu/machsuif

[33] VCG. Visualization for Compiler Graphs. University of Saarland, Germany
http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html

[34] NCI. National Compiler Infrastructure.
http://www.cs.virginia.edu/nci
http://nci.pgroup.com

[35] Holub, A. I., Compiler Design in C, Prentice-Hall of India, 2003

[36] Grune D., Bal, H. E., Jacobs, C. J. H., Langendoen, K. G., Modern Compiler Design, Wiley®-dreamtech publications India, 2003

[37] Pratt, T. W., Zelkowitz, M. V., Programming Languages Design and Implementation, 4th Edition, Prentice-Hall of India, 2005

[38] Beck, L. L., System Software: *An Introduction to Systems Programming*, 3rd Edition, Pearson Education, 2003

[39] Kernighan, B. W., Ritchie, D. M. The C Programming Language, 2th Edition, Pearson Education.

[40] Kelley, A., Pohl, I., A Book On C, 4th Edition, Pearson Education, 2001